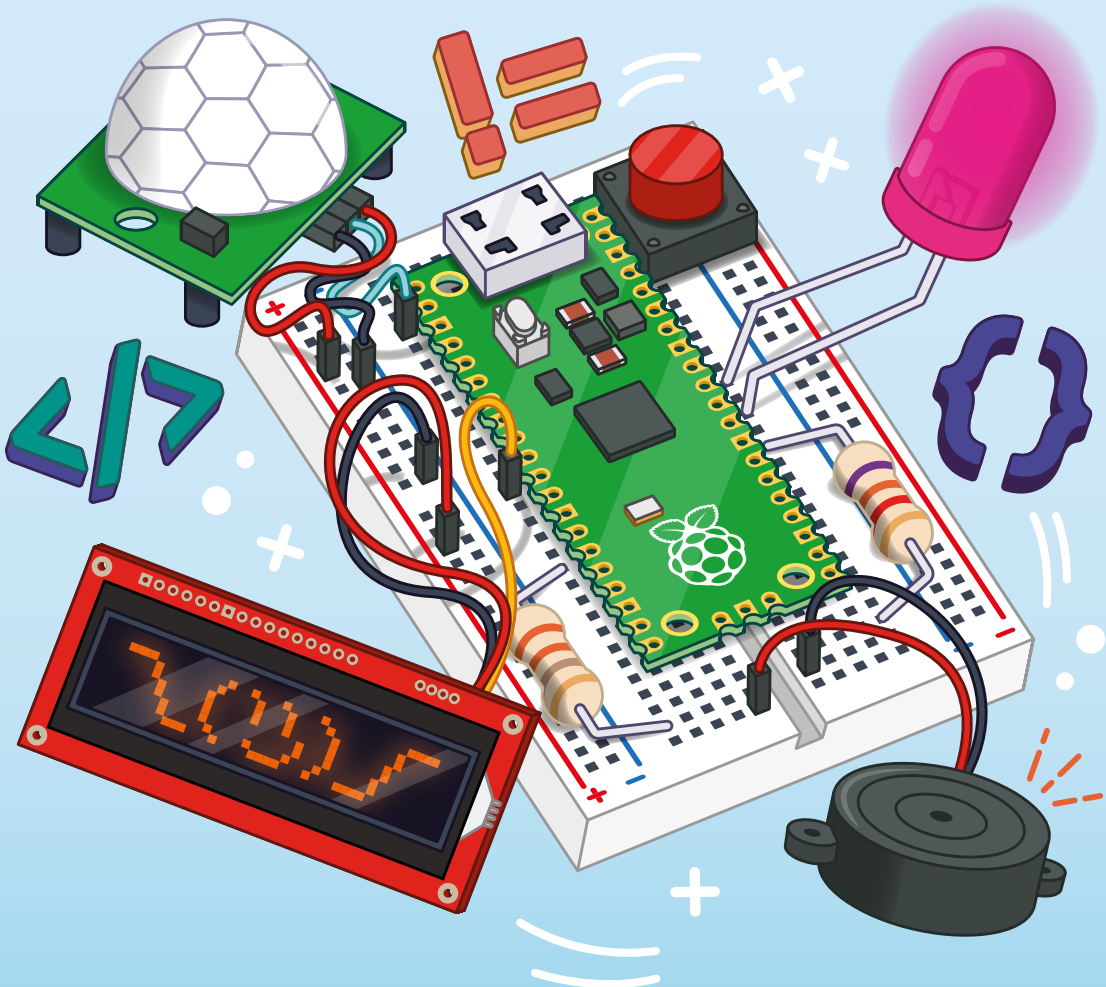
THE OFFICIAL **RASPBERRY PI PICO** GUIDE

Get started with MicroPython on Raspberry Pi Pico



by **Gareth Halfacree**
and **Ben Everard**

Raspberry Pi PRESS

Get started with MicroPython on Raspberry Pi Pico

First published in 2021 by Raspberry Pi Trading Ltd, Maurice Wilkes Building,
St. John's Innovation Park, Cowley Road, Cambridge, CB4 0DS

Publishing Director: Russell Barnes • Editor: Phil King • Sub Editor: Nicola King
Design: Critical Media • Illustrations: Sam Alder
CEO: Eben Upton

ISBN: 978-1-912047-86-4

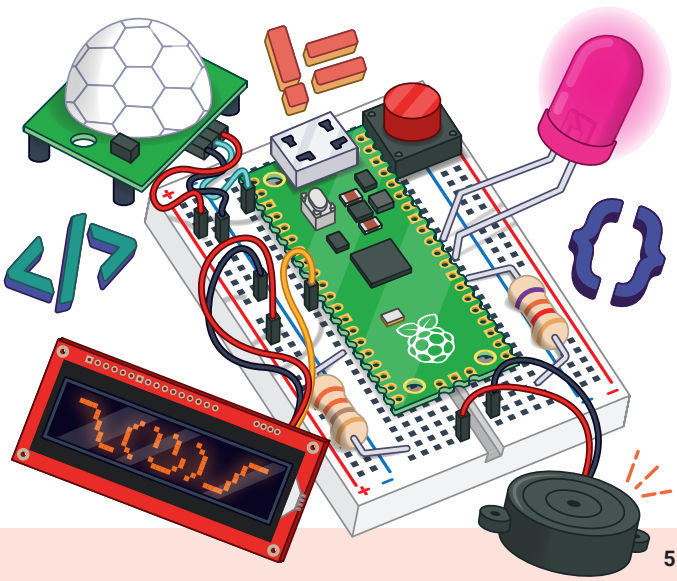
The publisher, and contributors accept no responsibility in respect of any omissions or errors relating to goods, products or services referred to or advertised in this book.
Except where otherwise noted, the content of this book is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported
(CC BY-NC-SA 3.0)

Welcome

You might think of computers as things you stick on your desk and type on, and this is certainly one type of computer, but it's not the only type. In this book, we're looking at microcontrollers – small processing units with a bit of memory that are good at controlling other hardware. You probably have lots of microcontrollers in your house already. There's a good chance your washing machine is controlled by a microcontroller; maybe your watch is; you might find one in your coffee machine or microwave. Of course, all these microcontrollers already have their programs and the manufacturers make it hard to change the software running on them.

A Raspberry Pi Pico, on the other hand, can be easily reprogrammed over a USB connection. In this book, we'll look at how to get started with this hardware, and how to work with other electronic components. By the end of the book, you'll know how to create your own programmable electronic contraptions. What you do with them is up to you.

Ben Everard



About the Authors

Gareth Halfacree is a freelance technology journalist, writer, and former system administrator in the education sector. With a passion for open-source software and hardware, he was an early adopter of the Raspberry Pi platform and has written several publications on its capabilities and flexibility. He can be found on Twitter as **@ghalfacree** or via his website at **freelance.halfacree.co.uk**.

Ben Everard is a geek who has stumbled into a career that lets him play with new hardware. As the editor of HackSpace magazine (**hsmag.cc**), he spends more time than he really should experimenting with the latest (and not-so-latest) DIY tech. He lives in Bristol with his wife and two daughters in a house that's slowly filling up with electronics equipment and 3D printers.

Contents

Chapter 1: Get to know your Raspberry Pi Pico **008**
Get fully acquainted with your powerful new microcontroller and learn how to attach pin headers and install MicroPython to program it

Chapter 2: Programming with MicroPython **020**
Connect a computer and start writing programs for your Raspberry Pi Pico using the MicroPython language

Chapter 3: Physical computing **034**
Learn about your Raspberry Pi Pico's pins and the electronic components you can connect and control

Chapter 4: Physical computing with Raspberry Pi Pico **044**
Start connecting basic electronic components to your Raspberry Pi Pico and writing programs to control and sense them

Chapter 5: Traffic light controller **058**
Create your own mini pedestrian crossing system using multiple LEDs and a push-button

Chapter 6: Reaction game **068**
Build a simple reaction timing game using an LED and push-buttons, for one or two players

Chapter 7: Burglar alarm **080**
Use a motion sensor to detect intruders and sound the alarm with a flashing light and siren

Chapter 8: Temperature gauge **092**
Use your Raspberry Pi Pico's built-in ADC to convert analogue inputs, and read its internal temperature sensor

Chapter 9: Data logger **104**
Turn Raspberry Pi Pico into a temperature data-logging device and untether it from the computer to make it fully portable

Chapter 10: Digital communication protocols: I2C and SPI **116**
Explore these two popular communication protocols and use them to display data on an LCD

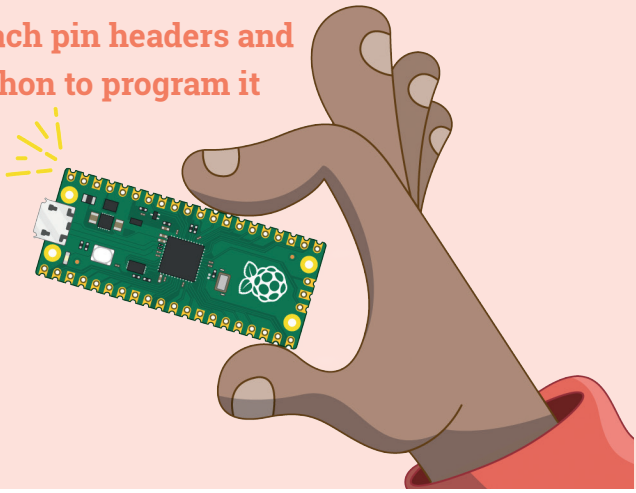
APPENDICES

Appendix A: Raspberry Pi Pico specifications **124**
Appendix B: Pinout guide **128**
Appendix C: Programmable IO **130**

Chapter 1

Get to know your Raspberry Pi Pico

Get fully acquainted with your powerful new microcontroller board and learn how to attach pin headers and install MicroPython to program it



Raspberry Pi Pico is a miniature marvel, putting the same technology that underpins everything from smart home systems to industrial factories in the palm of your hand. Whether you're looking to learn about the MicroPython programming language, take your first steps in physical computing, or want to build a hardware project, Raspberry Pi Pico – and its amazing community – will support you every step of the way.

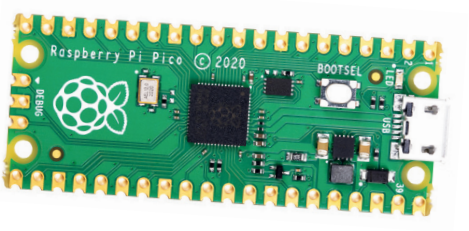
Raspberry Pi Pico is known as a *microcontroller development board*, meaning simply that it's a *printed circuit board* housing a special type of processor designed for physical computing: the *microcontroller*. The size of a stick of gum, Raspberry Pi Pico packs a surprising amount of power thanks to the chip at the centre of the board: a RP2040 microcontroller.

Raspberry Pi Pico isn't designed to replace Raspberry Pi, which is a different class of device known as a *single-board computer*. Whereas you might use your Raspberry Pi to play games, write stories, and browse the web, your Raspberry Pi Pico is designed for physical computing projects where it controls anything from LEDs and buttons to sensors, motors, and even other microcontrollers.

Throughout this book you'll be learning all about Raspberry Pi Pico, but the skills you learn will also apply to any other development board based around its RP2040 microcontroller – and even other devices, so long as they are compatible with the MicroPython programming language.

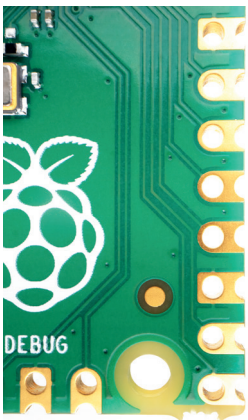
A guided tour of Raspberry Pi Pico

Raspberry Pi Pico – 'Pico' for short – is a lot smaller than even a Raspberry Pi Zero, the most compact of Raspberry Pi's single-board computer family. Despite this, it includes a lot of features – all accessible using the *pins* around the edge of the board.



▲ **Figure 1-1:** The top of the board

Figure 1-1 shows Raspberry Pi Pico as seen from above. If you look at the longer edges, you'll see gold-coloured sections which look a bit like little spacemen. These are the *pins* which provide the RP2040 microcontroller with connections to the outside world – known as *input/output (IO)*.

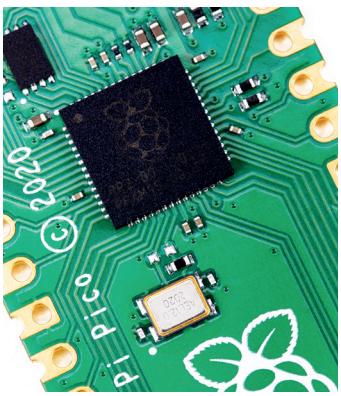


▲ **Figure 1-2:** Castellated

The pins on your Pico are very similar to the pins that make up the *general-purpose input/output (GPIO)* header on your Raspberry Pi – but while most Raspberry Pi single-board computers come with the physical metal pins already attached, your Pico doesn't. There's a good reason for that: look at the outer edge of the circuit board and you'll see it's bumpy, with little circular cut-outs (**Figure 1-2**).

These bumps create what is called a *castellated circuit board*, which can be attached on top of other circuit boards through *soldering* without ever having any physical metal pins fitted – which helps to keep the height down, making for a smaller finished project. If you buy an off-the-shelf gadget powered by Raspberry Pi Pico, it'll almost certainly be fitted using the castellations.

The holes just inwards from the bumps are for *2.54 mm male pin headers* – the same type of pins used on your Raspberry Pi's GPIO header. By soldering these in place pointing downwards, you can push your Pico into a *solderless breadboard* to make connecting and disconnecting new hardware as easy as possible – great for experiments!



▲ **Figure 1-3:** RP2040 chip

The chip at the centre of your Pico (**Figure 1-3**) is a an RP2040 microcontroller. This is a *custom integrated circuit (IC)*, designed and built specifically by Raspberry Pi's engineers to power your Pico and other microcontroller-based devices. If you hold it up to the light, you'll see a

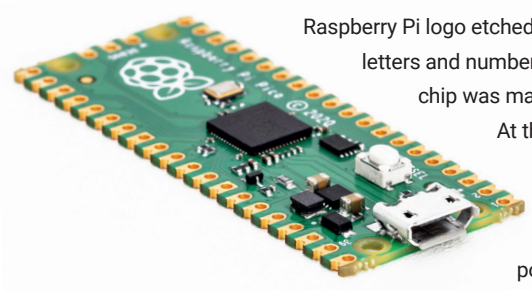


Figure 1-4: micro USB port

Raspberry Pi logo etched into the top of the chip along with a series of letters and numbers which let engineers track when and where the chip was made.

At the top of your Pico is a *micro USB port* (**Figure 1-4**). This provides power to make your Pico run, and also lets Pico talk to your Raspberry Pi or other computer via its USB port – which is how you'll load your programs onto your Pico. If you hold your Pico up and look at the micro USB port head-on, you'll see it's shaped so it's narrower at the bottom and wider at the top. Take a micro USB cable, and you'll see its connector is the same.

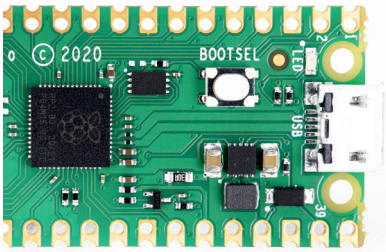


Figure 1-5: Boot selection switch

The micro USB cable will only go into the micro USB port on your Pico one way up. When you're connecting it, make sure to line the narrow and wide sides up the right way around – or you could damage your Pico trying to jam the micro USB cable in the wrong way up!

Just below the micro USB port is a small button marked *'BOOTSEL'* (**Figure 1-5**). 'BOOTSEL' is short for 'boot selection', which switches your Pico between two start-up modes when it's first switched on. You'll use the boot selection button later, as you get your Pico ready for programming with MicroPython.

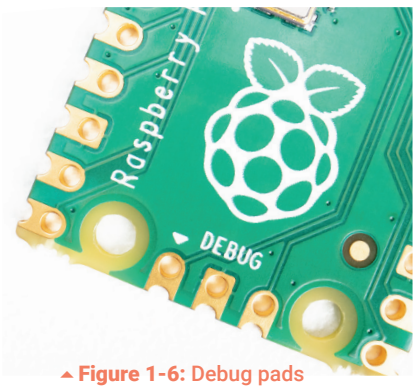


Figure 1-6: Debug pads

At the bottom of your Pico are three smaller gold pads with the word *'DEBUG'* above them (**Figure 1-6**). These are designed for *debugging*, or finding errors, in programs running on the Pico, using a special tool called a *debugger*. You won't be using the debug header in this book, but you may find it useful as you write larger and more complicated programs.

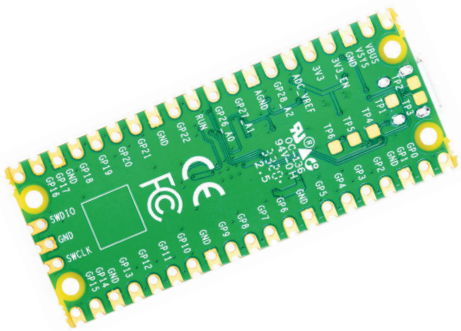


Figure 1-7: Labelled underside

Turn your Pico over, and you'll see the underside has writing on it (**Figure 1-7**). This is known as a *silk-screen layer*, and labels each of the pins with its core function. You'll see things like 'GP0' and 'GP1', 'GND', 'RUN', and '3V3'. If you ever forget which pin is which, these labels will tell you – but you won't be able to see them when the Pico is pushed into a breadboard, so you'll find full *pinout diagrams* printed in this book for easier reference.

You might have noticed that not all the labels line up with their pins: the small holes at the top and bottom of the board are *mounting holes*, designed to allow you to attach your Pico to projects more permanently using screws or nuts and bolts. Where the holes get in the way of the labelling, the labels are pushed further up or down the board: looking at the top-right, 'VBUS' is the first pin on the left, 'VSYS' the second, and 'GND' the third.

You'll also see some flat gold pads labelled with *'TP'* and a number. These are *test points*, and are designed for engineers to quickly check that a Raspberry Pi Pico is working after it has been assembled at the factory. Depending on the test pad, the engineer might use a tool called a *multimeter* or an *oscilloscope* to check that your Pico is working properly before it's packaged up and shipped to you.

Finally, you'll see a small sticker with a bar code on it. This holds the *serial number* of your Pico, as well as version information and the date it was manufactured.

Soldering the headers

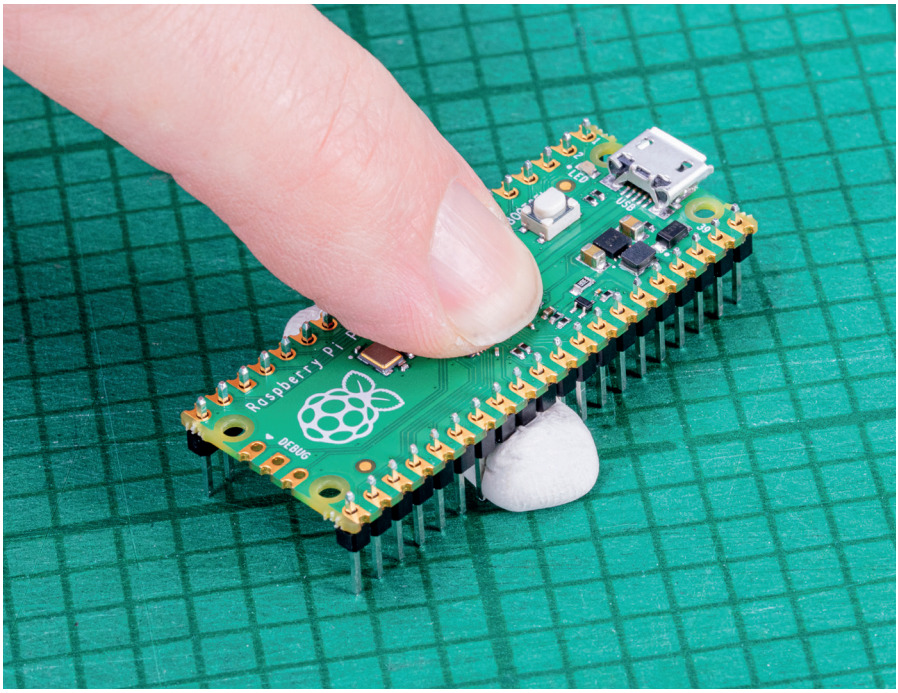
When you unpack your Raspberry Pi Pico, you'll notice that it is completely flat: there are no metal pins sticking out from the sides, like you'd find on the GPIO header of your Raspberry Pi. This is in case you wanted to use the castellations to attach your Pico to another circuit board, or to directly solder wires.

The easiest way to use your Pico, though, is to attach it to a breadboard – and for that, you'll need to attach pin headers. You'll need a soldering iron with a stand, some solder, a cleaning sponge, your Pico, and two 20-pin 2.54 mm male header strips. If you already have a solderless breadboard, you can use it to make the soldering process easier.

Sometimes 2.54 mm headers are provided in strips longer than 20 pins. If yours are longer, just count 20 pins in from one end and look at the plastic between the 20th and 21st pins: you'll see it has a small indentation at either side. This is a *break point*: put your thumbnails in the indentation with the headers in both your left and right hands and bend the strip. It will break cleanly, leaving you with a strip of exactly 20 pins. If the remaining header strip is longer than 20 pins, do the same again so you have two 20-pin strips.

WARNING

Soldering irons are not toys: they get very, very hot, and stay hot for a long time after they're unplugged. If you're a younger learner, make sure you have adult supervision; whether you're young or old, make sure that you put the iron in the stand when you're not using it and never ever touch the metal parts – even after it's unplugged.



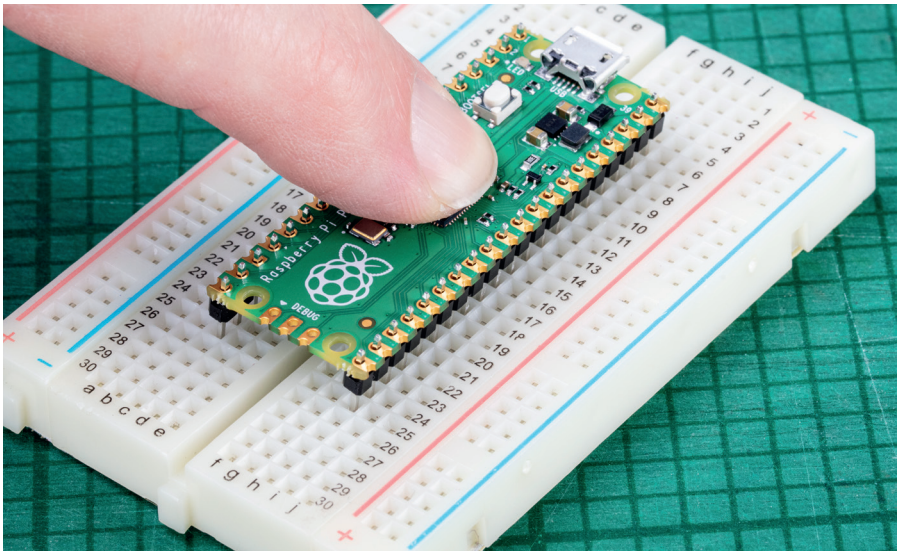
▲ **Figure 1-8:** You can hold the headers in place with sticky putty before soldering

Turn your Pico upside-down, so you can see the silk-screen pin numbers and test points on the bottom. Take one of the two header strips and push it gently into the pin holes on the left-hand side of your Pico. Make sure that it's properly inserted in the holes, and not just resting in the castellations, and that all 20 pins are in place, then take the other header and insert it into the right-hand side. When you've finished, the plastic blocks on the pins should be pushed up against your Pico's circuit board.

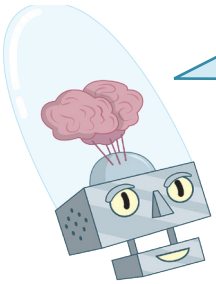
Pinch your Pico at the sides to hold both the circuit board and the two pin headers. Don't let go, or the headers will fall out! If you don't have a breadboard yet, you'll need some way to hold the headers in place while you're soldering – and don't use your fingers, or you'll burn them. You can hold the headers in place with small alligator clips, or a small blob of Blu Tack or other sticky putty (**Figure 1-8**). Solder one pin, then check the alignment: if the pins are at an angle, melt the solder as you carefully adjust them to get everything lined up.

If you do have a breadboard, simply turn your Pico upside down – remembering to keep the headers pinched – and push both the headers and your Pico into the holes on your breadboard. Keep pushing until your Pico is lying flat, with the plastic blocks on the pin headers sandwiched between your Pico and your breadboard (**Figure 1-9**).

Look at the top of your Pico: you'll see a small length of each pin is sticking up out of the pin holes. This is the part you're going to solder – which means heating up both the pins and the pads on the Pico and melting a small amount of a special metal, solder, onto them.



▲ **Figure 1-9:** Alternatively, use a breadboard to hold the headers in place for soldering



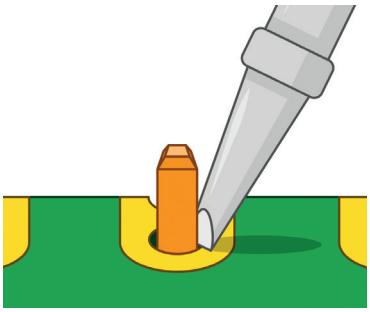
WARNING

Soldering is a great skill to learn, but it does take practice. Read the directions that follow carefully and in full before even turning your soldering iron on, and remember to take things slowly and carefully. Avoid using too much solder, too: it's easy to add more to a joint with too little solder, but can be harder to take excess solder away – especially if it's splashed over to other parts of your Pico.

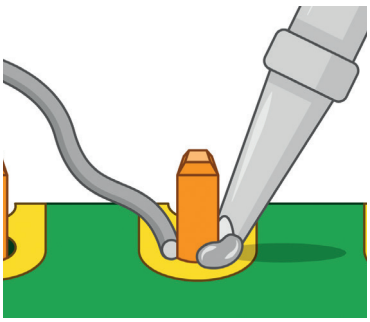
Put your soldering iron in its stand, making sure the metal tip isn't resting up against anything, and plug it in. It will take a few minutes for the tip of the iron to get hot; while you're waiting unroll a small length of solder – about twice as long as your index finger. You should be able to break the solder by pulling and twisting it; it's a very soft metal.

WARNING

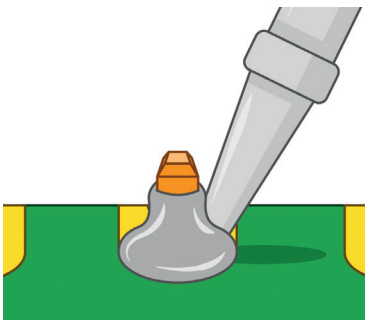
While modern solder is made without lead, it's still poisonous thanks to a special substance called *flux*. This is a corrosive gel which serves to burn dirt away from the joint as you're soldering. It won't harm you if you get it on your fingers, but it could make you ill if you were to eat it. Only handle the solder when you're actively using it, and always wash your hands afterwards – especially before you eat anything.



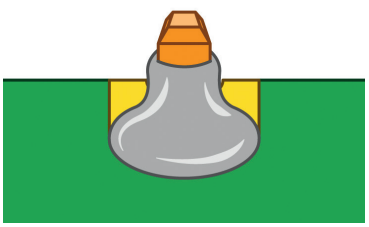
▲ **Figure 1-10:** Heat the pin and pad



▲ **Figure 1-11:** Add a little solder



▲ **Figure 1-12:** Now remove the iron



▲ **Figure 1-13:** A well-soldered pin

If your soldering stand has a cleaning sponge, take the sponge to the sink and put a little bit of cold water on it so it softens. Squeeze the excess water out of the sponge, so it's damp but not dripping, and put it back on the stand. If you're using a cleaner made of coiled brass wire, you don't need any water.

Pick up your soldering iron by the handle, making sure to keep the cable from catching on anything as you move it around. Hold it like a pencil, but make sure your fingers only ever touch the plastic or rubber handle area: the metal parts, even the shaft ahead of the actual iron tip, will be extremely hot and can burn you very quickly.

Before you begin soldering, clean the iron's tip: brush it along your sponge or coiled wire cleaner. Take your length of solder, holding it at one end, and push the other end onto the tip of your iron: it should quickly melt into a blob. If it doesn't, leave your soldering iron to heat up for longer – or try giving the tip another clean.

Putting a blob of solder on the tip is known as *tinning* the iron. The flux in the solder helps to burn off any dirt still on the end of the iron, and gets it ready. Wipe the iron on your sponge or cleaning wire again to clean off the excess solder; the tip should be left looking shiny and clean.

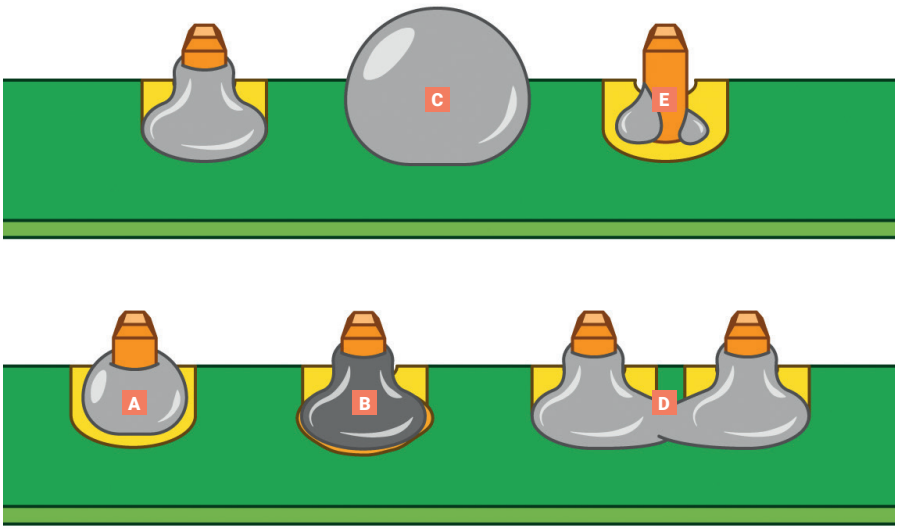
Put the iron back in the stand, where it should always be unless you're actively using it, and move your Pico so it's in front of you. Pick up the iron in one hand and the solder in the other. Press the tip of the iron against the pin closest to you, so that it's touching both the vertical metal pin and the gold-coloured pad on your Pico at the same time (**Figure 1-10**).

It's important that the pin and the pad are both heated up, so keep your iron pressed against both while you count to three. When you've reached three, still keeping the iron in place, press the end of your length of solder gently against both the pin and pad but on the opposite side to your iron tip (**Figure 1-11**). Just like when you tinned the tip, the solder should melt quickly and begin to flow.

The solder will flow around the pin and the pad, but no further: that's because your Pico's circuit board is coated in a layer called *solder resist* which keeps the solder where it needs to be. Make sure not to use too much solder: a little goes a long way.

Pull the remaining part of your solder away from the joint, making sure to keep the iron in place. If you pull the iron away first, the solder will harden and you won't be able to remove the piece in your hand; if that happens, just put the iron back in place to melt it again. Once the molten solder has spread around the pin and pad (**Figure 1-12**), which should only take a second or so, remove the soldering iron. Congratulations: you've soldered your first pin!

Clean the tip of your iron on your sponge or brass wire, and put it back in the stand. Pick up your Pico and look at your solder joint: it should fill the pad and rise up to meet the pin smoothly, looking a little like a volcano shape with the pin filling in the hole where the lava would be, as shown in **Figure 1-13**.



▲ **Figure 1-14:** Examples of soldering issues

If the solder is sticking to the pin but not sticking to the copper pad, as in example **A** in **Figure 1-14**, then the pad wasn't heated up enough. Don't worry, it's easily fixed: take your soldering iron and place it where the pad and pin meet, making sure that it's pressing against both this time. After a few seconds, the solder should reflow and make a good joint.

On the other hand, if the solder is too hot, it won't flow well and you'll get an overheated joint with some burnt flux (example **B** in **Figure 1-14**). This can be removed with a bit of careful scraping with the tip of a knife, or a toothbrush and a little isopropyl alcohol.

If the solder is entirely covering the pin, as in example **C** in **Figure 1-14**, you used too much. That's not necessarily going to cause a problem, though it doesn't look very attractive: so long as none of the solder is touching any of the pins around it, it should still work. If it is touching other pins (as in example **D** of **Figure 1-14**), you've created a *bridge* which will cause a *short circuit*.

Again, bridges are easy to fix. First, try reflowing the solder on the joint you were making; if that doesn't work, put your iron against the pin and pad at the other side of the bridge to flow some of it into the joint there. If there's far too much solder still, you'll need to remove the excess before you can use your Pico: you can buy *desoldering braid*, which you press against the molten solder to suck the excess up, or a *desoldering pump* to physically suck the molten solder up.

Another common mistake is too little solder: if you can still see copper pad, or there's a gap between the pin and the pad which isn't filled in with solder, you used too little (example **E** in **Figure 1-14**). Put the iron back on the pin and pad, count to three, and add a little more solder. Too little is always easier to fix than too much, so remember to take it easy with the solder!

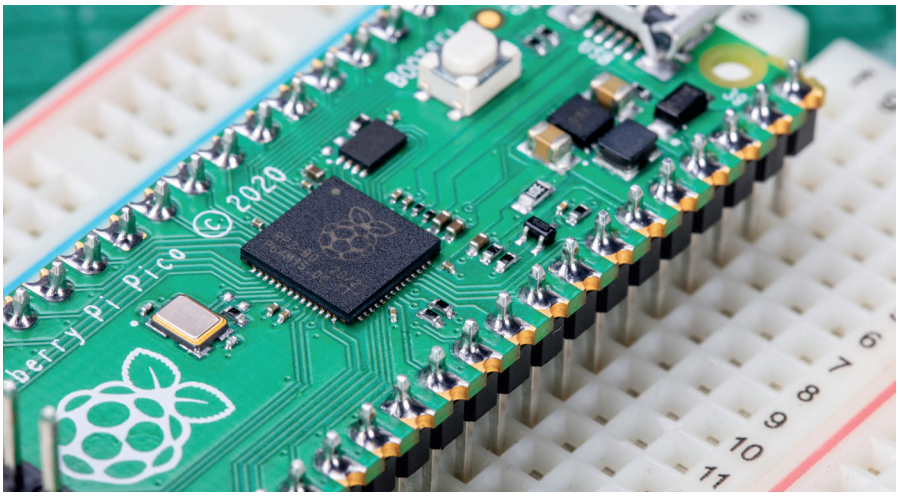
Once you're happy with the first pin, repeat the process for all 40 pins on your Pico – leaving the three-pin 'DEBUG' header at the bottom empty. Tip: solder the four corner pins first. Take your time, don't rush, and remember that mistakes can always be fixed. Remember to clean your iron's tip regularly during your soldering, too, and if you find things are getting difficult, melt some solder on it to re-tin the tip. Make sure to keep refreshing your length of solder, too: if it's too short and your fingers are too close to the soldering iron's tip, you can easily burn yourself.

When you're finished, and you've checked all the pins for good solder joints and to make sure they're not bridged to any nearby pins, clean and tin the iron's tip one last time before putting it back in the stand and unplugging it. Make sure to let the iron cool before you put it away: soldering irons can stay hot enough to burn you for a long time after they've been unplugged!

Finally, make sure to wash your hands – and celebrate your new skill as a soldering supremo!

Installing MicroPython

Now you've soldered the headers onto your Pico (**Figure 1-15**), there's only one thing left to do to get it ready: install MicroPython onto it. Start by plugging a micro USB cable into the micro USB port on your Pico, making sure that it's the right way up before gently pushing it home.



▲ **Figure 1-15:** All the pins correctly soldered

WARNING

To install MicroPython onto your Pico, you'll need to download it from the internet. If you don't have an internet connection on your Raspberry Pi, you'll need to connect your Pico to a system that does have an internet connection in order to finish setting it up. You'll only have to do this once: after MicroPython is installed, it will stay on your Pico unless you decide to replace it with something else in the future.

Hold down the 'BOOTSEL' button on the top of your Pico; then, while still holding it down, connect the other end of the micro USB cable to one of the USB ports on your Raspberry Pi or other computer. Count to three, then let go of the 'BOOTSEL' button.

After a few more seconds you should see your Pico appear as a removable drive – as though you'd connected a USB flash drive or external hard drive. On your Raspberry Pi you'll see a pop-up asking if you'd like to open the drive in the File Manager.

In the File Manager window, you'll see two files on your Pico (**Figure 1-16**): INDEX.HTM and INFO_UF2.TXT. The second file holds information about your Pico, such as the version of the bootloader it's currently running. The first file, INDEX.HTM, holds a link to the Pico website. Either click on this file, or point your web browser to raspberrypi.com/documentation/microcontrollers/.

When the web page opens, you'll see information about Raspberry Pi's microcontrollers including Pico and Pico W. Click on the MicroPython box to go to the page for this firmware. Scroll down this page to find the link for the version of MicroPython for your board (either Pico or Pico W). Click on the link to download the appropriate UF2 file.

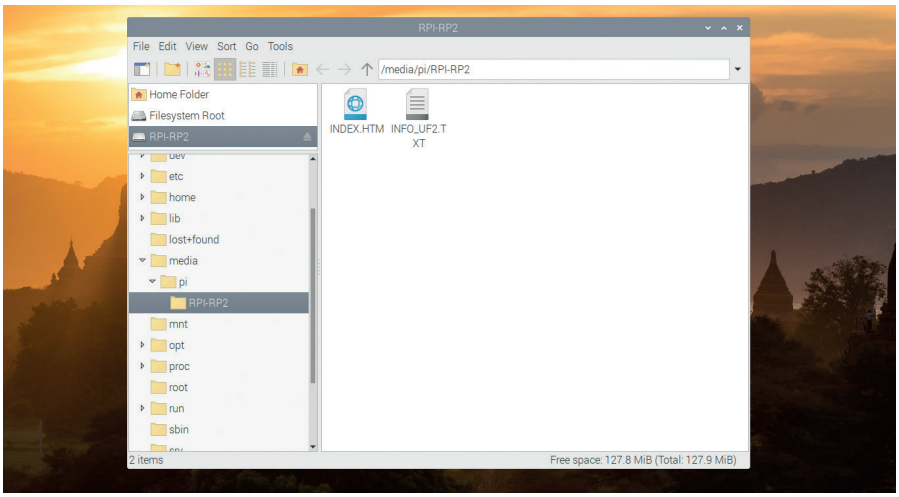
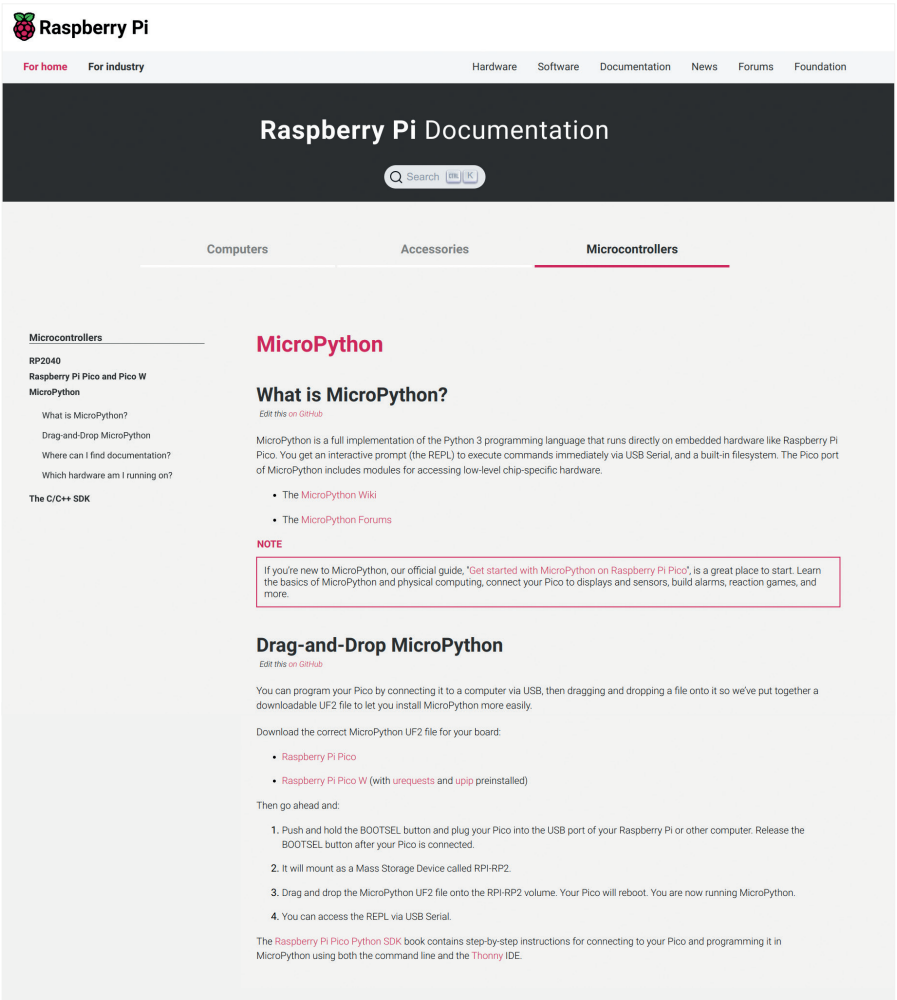


Figure 1-16: You'll see two files on your Raspberry Pi Pico

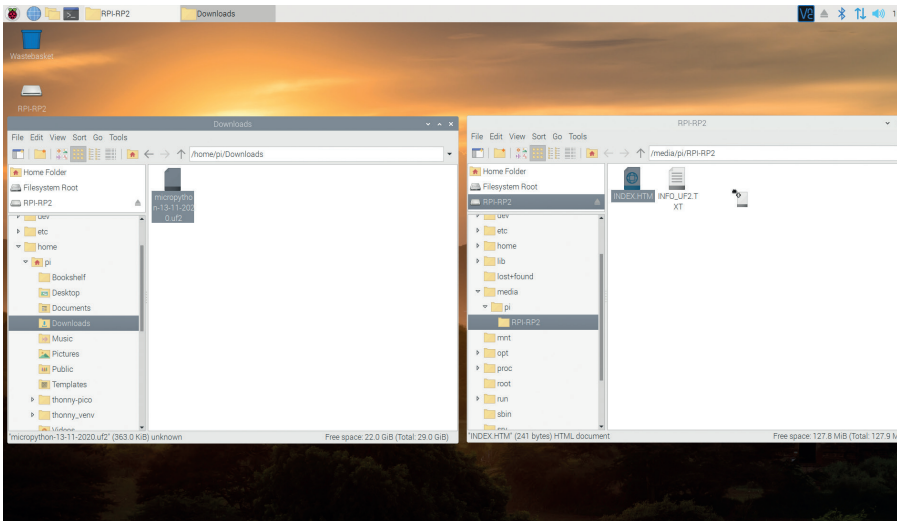

Raspberry Pi
For home
For industry
Hardware
Software
Documentation
News
Forums
Foundation
Raspberry Pi Documentation
Search
Computers
Accessories
Microcontrollers
Microcontrollers
RP2040
Raspberry Pi Pico and Pico W
MicroPython
What is MicroPython?
Drag-and-Drop MicroPython
Where can I find documentation?
Which hardware am I running on?
The C/C++ SDK
MicroPython
What is MicroPython?
Edit this on GitHub
MicroPython is a full implementation of the Python 3 programming language that runs directly on embedded hardware like Raspberry Pi Pico. You get an interactive prompt (the REPL) to execute commands immediately via USB Serial, and a built-in filesystem. The Pico port of MicroPython includes modules for accessing low-level chip-specific hardware.
The MicroPython Wiki
The MicroPython Forums
NOTE
If you're new to MicroPython, our official guide, "Get started with MicroPython on Raspberry Pi Pico", is a great place to start. Learn the basics of MicroPython and physical computing, connect your Pico to displays and sensors, build alarms, reaction games, and more.
Drag-and-Drop MicroPython
Edit this on GitHub
You can program your Pico by connecting it to a computer via USB, then dragging and dropping a file onto it so we've put together a downloadable UF2 file to let you install MicroPython more easily.
Download the correct MicroPython UF2 file for your board:
Raspberry Pi Pico
Raspberry Pi Pico W (with urequests and upip preinstalled)
Then go ahead and:
1. Push and hold the BOOTSEL button and plug your Pico into the USB port of your Raspberry Pi or other computer. Release the BOOTSEL button after your Pico is connected.
2. It will mount as a Mass Storage Device called RPI-RP2.
3. Drag and drop the MicroPython UF2 file onto the RPI-RP2 volume. Your Pico will reboot. You are now running MicroPython.
4. You can access the REPL via USB Serial.
The Raspberry Pi Pico Python SDK book contains step-by-step instructions for connecting to your Pico and programming it in MicroPython using both the command line and the Thonny IDE.

▲ **Figure 1-17:** Click the button to download the MicroPython firmware

Open a new File Manager window by clicking on the raspberry icon menu, going to Accessories, and clicking on File Manager. Find your **Downloads** folder using the list of folders to the left of the File Manager window. You may have to scroll the list to find it, depending on how many folders you have on your Raspberry Pi. Open the **Downloads** folder and find the file you just downloaded – it will be called 'micropython' followed by a date and the extension 'uf2'.

Click and hold the mouse button on the UF2 file then drag it to the other File Manager window open on your Pico's removable storage drive. Hover it over the window and let go of the mouse button to drop the file onto your Pico (**Figure 1-18**).

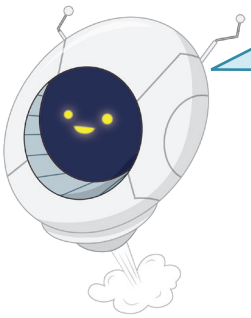
After a few seconds you'll see your Pico disappear from File Manager, and you may also see a warning that a drive was removed without being ejected: don't worry, that's supposed to happen!



▲ **Figure 1-18:** Drag the MicroPython firmware file to your Raspberry Pi Pico

When you dragged the MicroPython firmware file onto your Pico, you told it to *flash* the firmware onto its internal storage. To do that, your Pico switches out of the special mode you put it in with the 'BOOTSEL' button, flashes the new firmware, and then loads it – meaning that your Pico is now running MicroPython.

Congratulations: you're now ready to get started with MicroPython on your Raspberry Pi Pico!



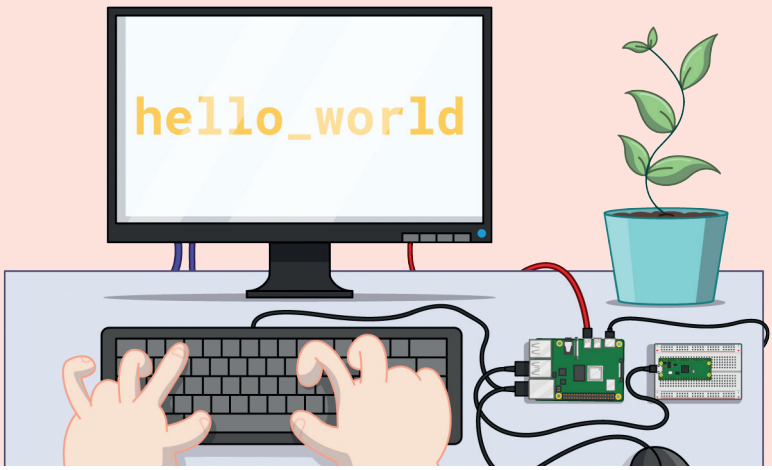
FURTHER READING

The webpage linked from **INDEX.HTM** isn't just a place to download MicroPython; it also includes a wealth of additional resources. Click on the tabs and scroll to access guides, projects, and the *data book* collection – a bookshelf of detailed technical documentation covering everything from the inner workings of the RP2040 microcontroller which powers your Pico to programming in both the Python and C/C++ languages.

Chapter 2

Programming with MicroPython

Connect a computer and start writing programs for your Raspberry Pi Pico using the MicroPython language



Since its launch in 1991, the Python programming language – named after the famous comedy troupe Monty Python, rather than the snake – has grown to become one of the most popular in the world. Its popularity, though, doesn't mean there aren't improvements that could be made – particularly if you're working with a microcontroller.

The Python programming language was developed for computer systems like desktops, laptops, and servers. Microcontroller boards like Raspberry Pi Pico are smaller, simpler, and with considerably less memory – meaning they can't run the same Python language as their bigger counterparts.

That's where MicroPython comes in. Originally developed by Damien George and first released in 2014, MicroPython is a Python-compatible programming language developed specifically for microcontrollers. It includes many of the features of mainstream Python, while adding a range of new ones designed to take advantage of the facilities available on Raspberry Pi Pico and other microcontroller boards.

If you've programmed with Python before, you'll find MicroPython immediately familiar. If not, don't worry: it's a friendly language to learn!

Figure 2-1: The Thonny Python IDE

Introducing the Thonny Python IDE

A Toolbar – The toolbar offers an icon-based quick-access system to commonly used program functions – like saving, loading, and running programs.

B Script Area – The script area is where your Python programs are written. It is split into a main area for your program and a small side margin for showing line numbers.

C Python Shell – The Python Shell allows you to type individual instructions which are run as soon as you press the **ENTER** key, and also provides information about running programs. This is also known as *REPL*, for 'read, evaluate, print, and loop.'

D Interpreter – The bottom-right of the Thonny window shows, and lets you change, the current Python *interpreter* – the version of Python used to run your programs.

Connecting Thonny to Pico

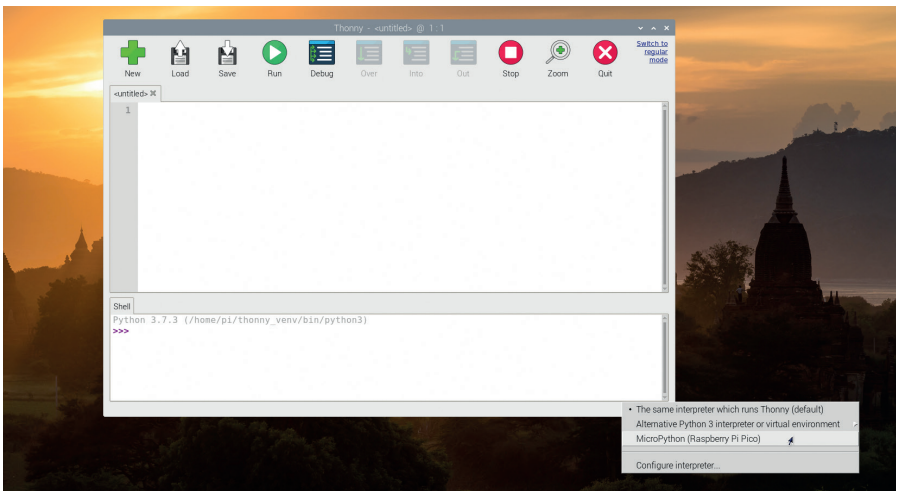
Before you can start to program Pico with MicroPython, you'll need to set up what is called an *integrated development environment (IDE)*. Thonny, a popular IDE for Python and MicroPython, comes preloaded on Raspberry Pi OS; if you're using Pico with a different Linux distribution, Windows, or macOS, open your web browser, visit **thonny.org**, and click the download link at the top of the page to download the Thonny and Python bundle installer for your operating system.

As an integrated development environment, Thonny gathers together, *or integrates*, all the different tools you need to write, or *develop*, software into a single user interface, or *environment*. There are many different IDEs: some allow you to develop in multiple different programming languages while others, like Thonny, focus on a single language.

Begin by loading Thonny: on Raspberry Pi OS, you can load it by clicking on the raspberry menu at the top-left on your screen, moving the mouse to the Programming section, and clicking on Thonny. If you haven't already done so, take your Pico and connect a micro USB cable between it and one of Raspberry Pi's USB ports – it doesn't matter which one.

With your Pico connected to your Raspberry Pi, click on the word 'Python' followed by a version number at the bottom-right of the Thonny window – the area marked as **D** in **Figure 2-1**. This shows your current *interpreter*, which is responsible for taking the instructions you type and turning them into code that the computer, or microcontroller, can understand and run. Normally the interpreter is the copy of Python running on Raspberry Pi, but it needs to be changed in order to run your programs in MicroPython on your Pico.

In the list that appears, look for 'MicroPython (Raspberry Pi Pico)' (**Figure 2-2**) and click on it. If you can't see it in the list, double-check that your Pico is properly plugged in to the micro USB cable and that the micro USB cable is properly plugged in to your Raspberry Pi or other computer.



▲ **Figure 2-2:** Choosing a Python interpreter

PYTHON PROFESSIONALS

If you've worked through the Python chapter in *The Official Raspberry Pi Beginner's Guide*, much of what you'll read in this chapter will be very familiar. Still, work through the first couple of examples, to get used to the differences in running programs in the Python interpreter on your Raspberry Pi and in the MicroPython interpreter on your Pico, then feel free to skip to the next chapter.

NO RASPBERRY PI PICO OPTION?

The Raspberry Pi Pico interpreter is only available in the latest version of Thonny. If you're running an older version and can't update it, look for 'MicroPython (generic)' instead. If your version of Thonny is older still and has no interpreter option at the bottom-right of the window and you can't update it, click 'Switch to regular mode' at the top-right, restart Thonny, click the Run menu, and click 'Select interpreter.' Click the drop-down arrow next to 'The same interpreter that runs Thonny (default)', click on 'MicroPython (generic)' in the list, then click on the drop-down arrow next to 'Port' and click on 'Board in FS mode' in that list before clicking OK to confirm your changes.

Look at the Python Shell at the bottom of the Thonny window: you'll see that it now reads 'MicroPython' and tells you that it's running on 'Raspberry Pi Pico'. Congratulations: you're ready to start programming.

Your first MicroPython program: Hello, World!

To start writing your first program, click on the Python Shell area at the bottom of the Thonny window, just to the right of the bottom '>>>' symbols, and type the following instruction before pressing the **ENTER** key.

```
print("Hello, World!")
```

When you press **ENTER**, you'll see that your program begins to run instantly: Python will respond, in the same Shell area, with the message 'Hello, World!' (**Figure 2-3**), just as you asked. That's because the Shell is a direct line to the MicroPython interpreter running on your Pico, whose

SYNTAX ERROR

If your program doesn't run but instead prints a 'syntax error' message to the Shell area, there's a mistake somewhere in what you've written. MicroPython needs its instructions to be written in a very specific way: miss a bracket or a quotation mark, spell 'print' wrong or give it a capital P, or add extra symbols somewhere in the instruction and it won't run. Try typing the instruction again, and make sure it matches the version in this book before pressing the **ENTER** key!

▲ **Figure 2-3:** MicroPython prints the 'Hello, World!' message in the Shell area

job it is to look at your instructions and interpret what they mean. This is known as *interactive mode*, and you can think of it like a face-to-face conversation with someone: as soon as you finish what you're saying, the other person will respond, then wait for whatever you say next.

Programming your Pico using the Shell is a little like having a telephone conversation: when you press the **ENTER** key, your instruction is sent through the micro USB cable to the MicroPython interpreter running on your Pico; the interpreter looks at your instruction, does whatever it is told, then sends the result back through the micro USB cable to Thonny.

You don't have to program in interactive mode via the Shell, though. Click on the script area in the middle of the Thonny window, then type your program again:

```
print("Hello, World!")
```

When you press the **ENTER** key this time, nothing happens – except that you get a new, blank line in the script area. To make this version of your program work, you'll have to click the Run icon in the Thonny toolbar or click the Run menu followed by 'Run current script'.

Click the Run icon now: you'll be asked whether you want to save your program to 'This computer', meaning your Raspberry Pi, or 'MicroPython device', meaning your Pico, as in **Figure 2-4**. Click 'MicroPython device', then type a descriptive name like **Hello_World.py** and click the OK button.

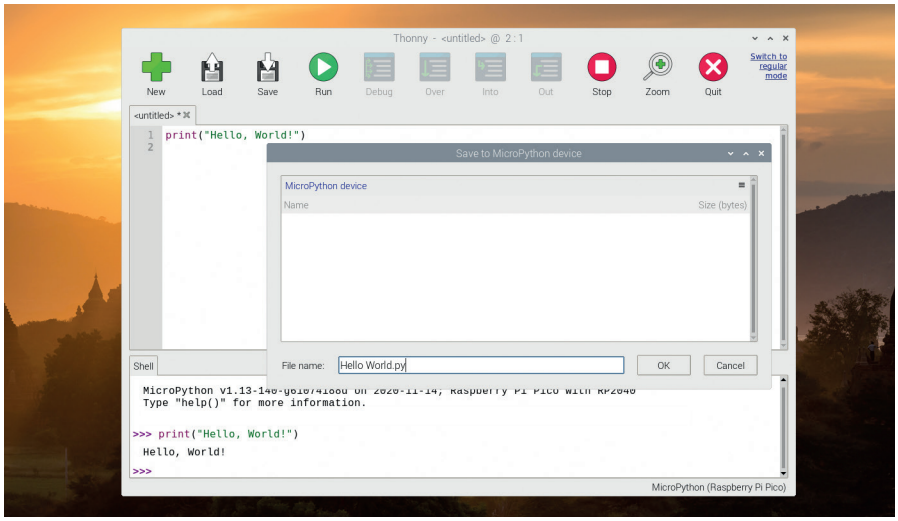
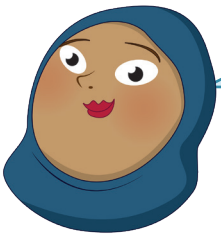


Figure 2-4: Saving a program to Pico



FILE NAMES

When saving MicroPython files to your Pico, always remember to type the *file extension*: a full-stop followed by the letters 'p' and 'y' – for 'Python' – at the end of the file. This helps you remember that each file is a program, and stops them getting mixed up with any other files you may save on your Pico.

You can use almost any name you like for your programs, but try to make it descriptive of what the program does – and don't call it **boot.py** or **main.py**, as these are special file names you'll learn more about later in the book.

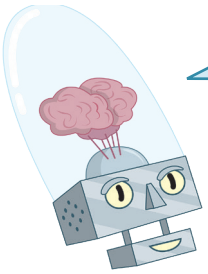
Once your program has saved, it will automatically run on your Pico. You'll see two messages appear in the Shell area at the bottom of the Thonny window:

```
>>> %Run -c $EDITOR_CONTENT
Hello, World!
```

The first of these lines is an instruction from Thonny telling the MicroPython interpreter on your Pico to run the contents of the script area – the 'EDITOR_CONTENT'. The second is the output of the program – the message you told MicroPython to print. Congratulations: now you've written two MicroPython programs, in interactive and script mode, and successfully run them on your Pico!

There's just one more piece to the puzzle: loading your program again. Close Thonny by pressing the X at the top-right of the window, then load it again. This time instead of writing a new program, click the Open icon in the Thonny toolbar. Like when you saved your program, you'll be asked whether you want to save it to 'This computer' or your 'MicroPython device' – click 'MicroPython device' and you'll see a list of all the programs you've saved to your Pico.

Find **Hello_World.py** in the list – if your Pico is new, it will be the only file there – and click on it to select it, then click OK. Your program will load into Thonny, ready to be edited or for you to run it again.



A PICO FULL OF PROGRAMS

When you tell Thonny to save your program on the MicroPython device, it means that the programs are stored on the Pico itself. If you unplug your Pico and take it to your friend's house, a school event, or a coding club and plug it into one of their computers, your programs will still be where you saved them – on your very own Pico.

CHALLENGE: NEW MESSAGE

Can you change the message the Python program prints as its output? If you wanted to add more messages, would you use interactive mode or script mode? What happens if you remove the brackets or the quotation marks from the program and then try to run it again?

Next steps: loops and code indentation

A MicroPython program, just as with a standard Python program, normally runs top-to-bottom: it goes through each line in turn, running it through the interpreter before moving on to the next, just as if you were typing them line-by-line into the Shell.

A program that just runs through a list of instructions line-by-line wouldn't be very clever, though – so MicroPython, just like Python, has its own way of controlling the sequence in which its programs run: *indentation*.

Create a new program by clicking on the New icon in the Thonny toolbar. You won't lose your existing program; instead, Thonny will create a new tab above the script area. Start your program by typing in the following two lines:

```
print("Loop starting!")
for i in range(10):
```

The first line prints a simple message to the Shell, just like your Hello World program. The second begins a *definite loop*, which will repeat – loop – one or more instructions a set number of times. A *variable*, **i**, is assigned to the loop and given a series of numbers – the **range** instruction, which is told to start at the number 0 and work upwards towards, but never reaching, the number 10 – to count. The colon symbol (**:**) tells MicroPython that the loop itself begins on the next line.

Variables are powerful tools: as their name suggests, variables are values which can change – or vary – over time and under the control of the program. At its most simple, a variable has two aspects: its name, and the *data* it stores. In the case of your loop, the variable's name is 'i' and its data is set by the **range** instruction – starting at 0 and increasing by 1 each time the loop finishes and begins afresh.

To actually include a line of code in the loop, it has to be *indented* – moved in from the left-hand side of the script area. The next line starts with four blank spaces, which Thonny will have added automatically when you pressed **ENTER** after line 2. Type it in now:

```
    print("Loop number", i)
```

The four blank spaces push this line inwards compared to the other lines in your program. This indentation is how MicroPython tells the difference between instructions outside the loop and instructions inside the loop: the indented code, forming the inside of the loop, is known as being *nested*.

You'll notice that when you pressed **ENTER** at the end of the third line, Thonny automatically indented the next line – assuming it would be part of the loop. To remove this indentation, just press the **BACKSPACE** key once before typing the fourth line:

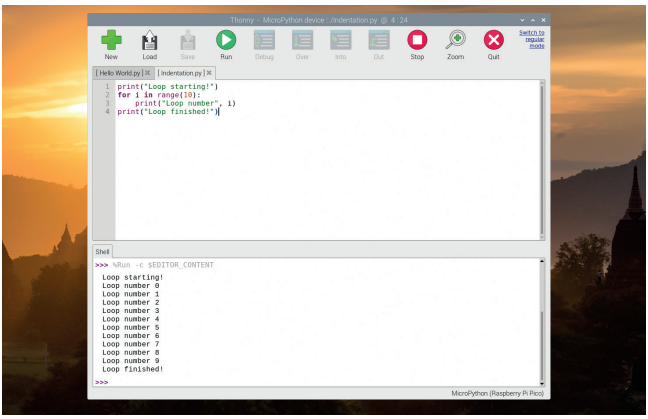
```
print("Loop finished!")
```

Your four-line program is now complete. The first line sits outside the loop, and will only run once; the second line sets up the loop; the third sits inside the loop and will run once for each time the loop loops; and the fourth line sits outside the loop once again.

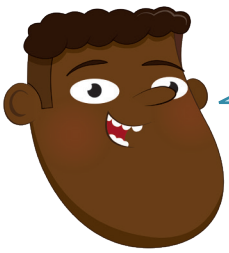
```
print("Loop starting!")
for i in range(10):
    print("Loop number", i)
print("Loop finished!")
```

Click the Run icon, choose to save the program on your Pico by clicking 'MicroPython device', and call it **Indentation.py**. The program will run as soon as it is saved: look at the Shell area for its output (**Figure 2-5**, overleaf).

```
Loop starting!
Loop number 0
Loop number 1
Loop number 2
Loop number 3
Loop number 4
Loop number 5
Loop number 6
Loop number 7
Loop number 8
Loop number 9
Loop finished!
```



▲ **Figure 2-5:** Executing a loop



COUNT FROM ZERO

Python is a *zero-indexed* language – meaning it starts counting from 0, not from 1. This is why your program prints the numbers 0 through 9 rather than 1 through 10. If you wanted to, you could change this behaviour by switching the `range(10)` instruction to `range(1, 11)` – or any other numbers you like.

Indentation is one of the most common reasons for a program to not work as you expected. When looking for problems in a program, a process known as *debugging*, always double-check the indentation – especially when you begin nesting loops within loops.

MicroPython also supports *infinite* loops, which run without end. To change your program from a definite loop to an infinite loop, edit line 2 to read:

```
while True:
```

Since we'll no longer be using the variable `i`, change line 3 to read:

```
    print("Loop running!")
```

To avoid the program running too quickly, we'll also add a short time delay by importing the utime library at the start and adding a one-second `sleep` delay to the loop (you'll learn more about this library in later chapters). Your program should now look like this:

```
import utime
print("Loop starting!")
while True:
```

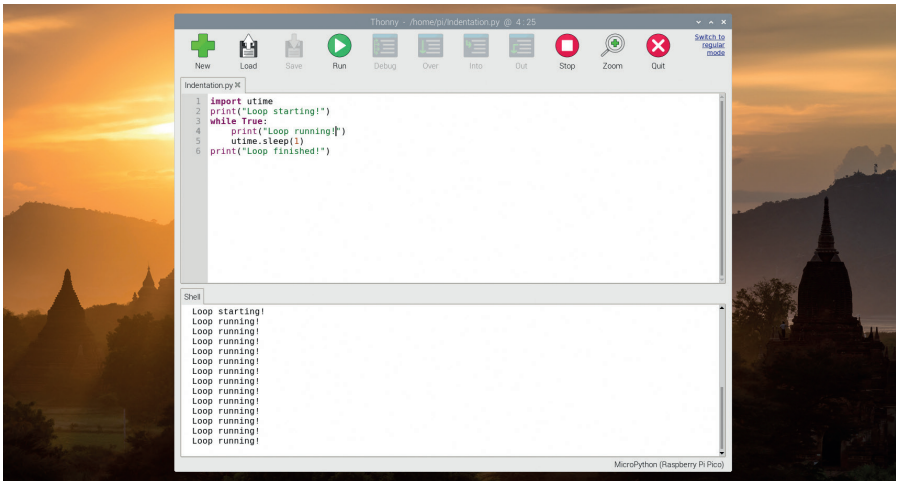
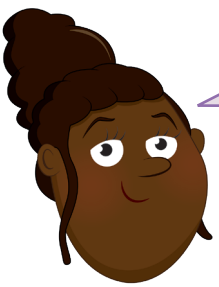


Figure 2-6: An infinite loop, which keeps going until you stop the program

```
    print("Loop running!")
    utime.sleep(1)
print("Loop finished!")
```

Click the Run icon again, and you'll see the 'Loop starting!' message followed by a never-ending string of 'Loop running!' messages (**Figure 2-6**). The 'Loop finished!' message will never print, because the loop has no end: every time Python has finished printing the 'Loop running!' message, it goes back to the beginning of the loop and prints it again. Click the Stop icon on the Thonny toolbar to tell the program to stop what it's doing – known as *interrupting* the program – and to restart the MicroPython interpreter. You'll see a message appear in the Shell area and the program will stop, without ever reaching line 6.

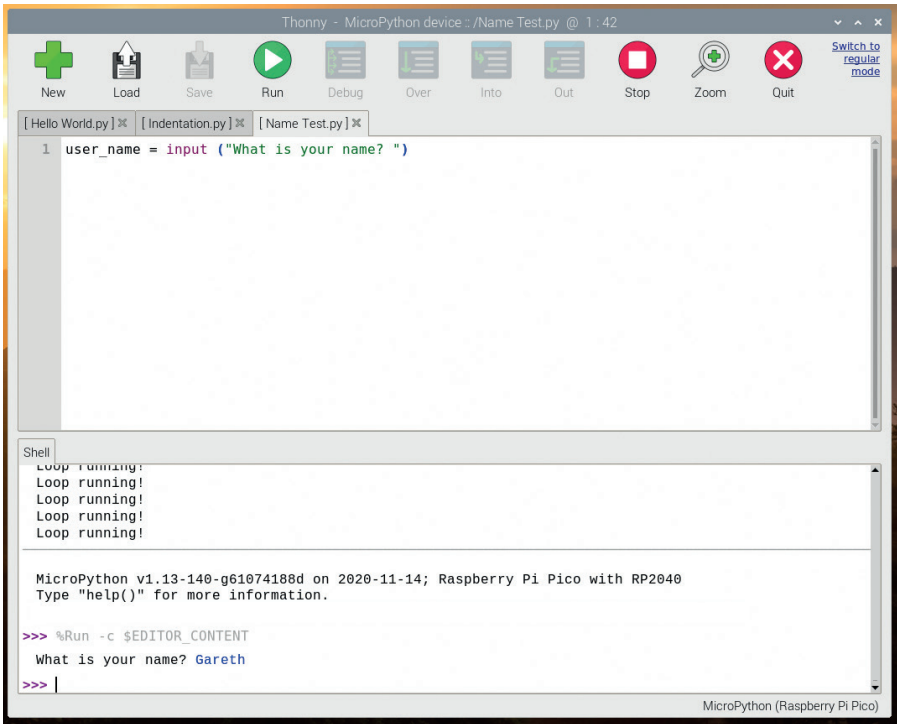


CHALLENGE: LOOP THE LOOP

Can you change the loop back into a definite loop again? Can you add a second definite loop to the program? How would you add a loop within a loop, and how would you expect that to work?

Conditionals and variables

Variables in MicroPython, as in all programming languages, exist for more than just controlling loops. Start a new program by clicking the New icon on the Thonny toolbar, then type the following into the script area:



▲ **Figure 2-7:** The `input` function lets you ask a user for some text input

```
user_name = input ("What is your name? ")
```

Click the Run icon, choose to save your program on your Pico by clicking 'MicroPython device,' give it the name **Name_Test.py**, and watch what happens in the Shell area: you'll be asked for your name. Type your name into the Shell area, followed by **ENTER**. Because that's the only instruction in your program, nothing else will happen (**Figure 2-7**, overleaf). If you want to actually do anything with the data you've placed into the variable, you'll need more lines in your program.

USING = AND ==

The key to using variables is to learn the difference between = and ==. Remember: = means 'make this variable equal to this value', while == means 'check to see if the variable is equal to this value'. Mixing them up is a sure way to end up with a program that doesn't work!

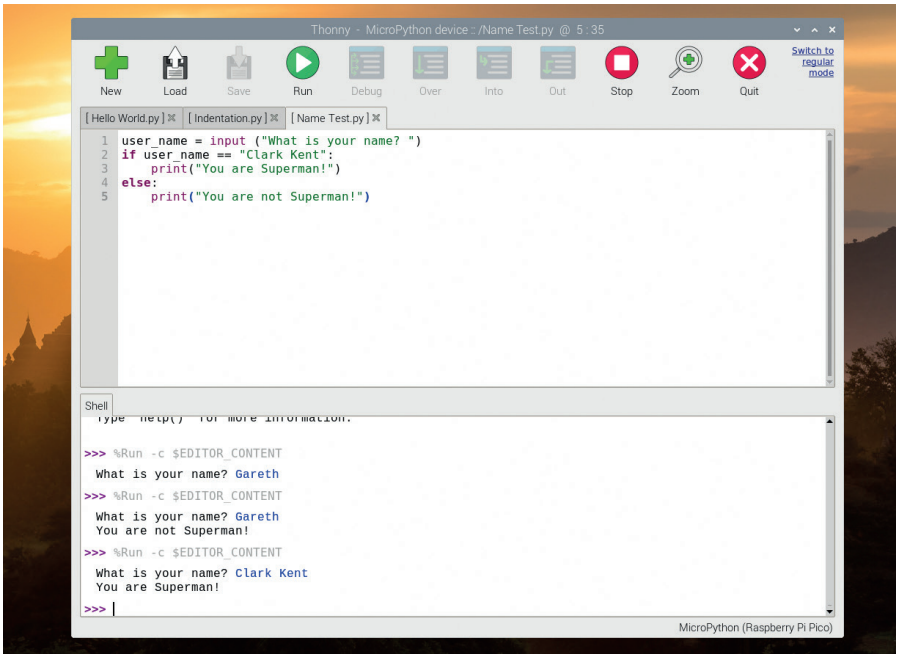


Figure 2-8: Shouldn't you be out saving the world?

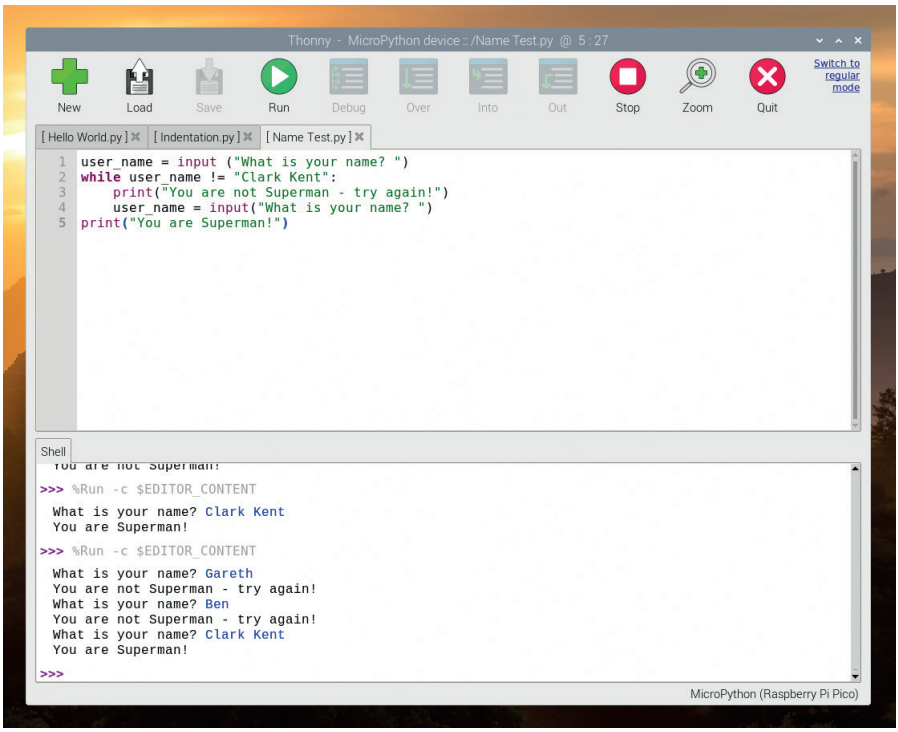
To make your program do something useful with the name, add a *conditional statement* by typing the following from line 2 onwards:

```
if user_name == "Clark Kent":
    print("You are Superman!")
else:
    print("You are not Superman!")
```

Remember that when Thonny sees that your code needs to be indented, it will do so automatically – but it doesn't know when your code needs to stop being indented, so you'll have to delete the spaces yourself.

Click the Run icon and type your name into the Shell area. Unless your name happens to be Clark Kent, you'll see the message 'You are not Superman!'. Click Run again, and this time type in the name 'Clark Kent' – making sure to write it exactly as in the program, with a capital C and K. This time, the program recognises that you are, in fact, Superman (**Figure 2-8**).

The `==` symbols tell Python to do a direct comparison, looking to see if the variable `user_name` matches the text – known as a *string* – in your program. If you're working with numbers, there are other comparisons you can make: `>` to see if a number is greater than another number, `<` to see if it's less than, `>=` to see if it's greater than or equal to, `<=` to see if it's less than or equal to.

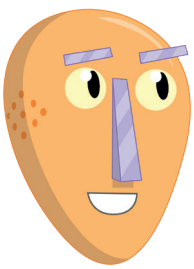


▲ **Figure 2-9:** The program will keep asking for your name until you say it's 'Clark Kent'

There's also `!=`, which means not equal to – it's the exact opposite of `==`. These symbols are technically known as *comparison operators*.

Comparison operators can also be used in loops. Delete lines 2 through 5, then type the following in their place:

```
while user_name != "Clark Kent":
    print("You are not Superman - try again!")
    user_name = input("What is your name? ")
print("You are Superman!")
```



CHALLENGE: ADD MORE QUESTIONS

Can you change the program to ask more than one question, storing the answers in multiple variables? Can you make a program which uses conditionals and comparison operators to print whether a number typed in by the user is higher or lower than 5?

Click the Run icon again. This time, rather than quitting, the program will keep asking for your name until it confirms that you are Superman (**Figure 2-9**) – sort of like a very simple password. To get out of the loop, either type 'Clark Kent' into the script area or click the Stop icon on the Thonny toolbar. Congratulations: you now know how to use conditionals and comparison operators!

Chapter 3
Physical computing

Learn about your Raspberry Pi Pico's pins and the electronic components you can connect and control

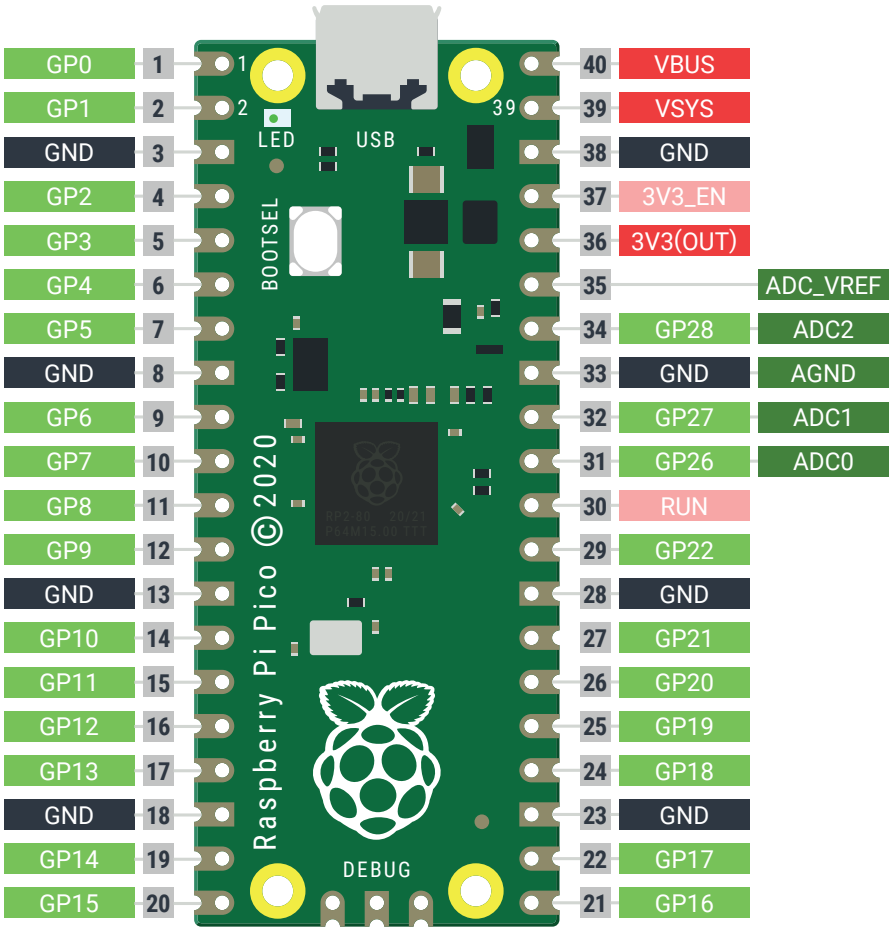
When people think of 'programming' or 'coding', they're usually – and naturally – thinking about software. Coding can be about more than just software, though: it can affect the real world through hardware. This is known as *physical computing*. As the name suggests, physical computing is all about controlling things in the real world with your programs: hardware, rather than software. When you set the program on your washing machine, change the temperature on your programmable thermostat, or press a button at traffic lights to cross the road safely, you're using physical computing.

These devices are typically controlled by a microcontroller very much like the one on your Raspberry Pi Pico – and it's entirely possible for you to create your own control systems by learning to take advantage of your Pico's capabilities, just as easily as you learned to write software that runs on your Pico.

Your Pico's pins

Your Pico talks to hardware through a series of pins along both its edges. Most of these pins work as a *general-purpose input/output (GPIO)* pin, meaning they can be programmed to act as either an input or an output and have no fixed purpose of their own. Some pins have extra features and alternative modes for communicating with more complicated hardware; others have a fixed purpose, providing connections for things like power.

Raspberry Pi Pico's 40 pins are labelled on the underside of the board, with three also labelled with their numbers on the top of the board: Pin 1, Pin 2, and Pin 39. These top labels help you remember how the numbering works: Pin 1 is at the top-left as you look at the board from above with the micro USB port to the upper side, Pin 19 is the bottom-left, Pin 20 the bottom-right, and Pin 39 one below the top-right with the unlabelled Pin 40 above it.



▲ **Figure 3-1:** The Raspberry Pi Pico's pins, seen from the top of the board

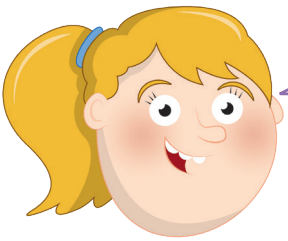
Rather than use the physical pin numbers, though, it's more common to refer to the pins by the functions on each (see **Figure 3-1**). There are several categories of pin types, each of which has a particular function:

3V3	3.3 volts power	A source of 3.3 V power, the same voltage your Pico runs at internally, generated from the VSYS input. This power supply can be switched on and off using the 3V3_EN pin above it, which also switches your Pico off.
VSYS	~2-5 volts power	A pin directly connected to your Pico's internal power supply, which cannot be switched off without also switching Pico off.
VBUS	5 volts power	A source of 5 V power taken from your Pico's micro USB port, and used to power hardware which needs more than 3.3 V.
GND	0 volts ground	A ground connection, used to complete a circuit connected to a power source. Several of these pins are dotted around your Pico to make wiring easier.
GP*xx*	General-purpose input/output pin number '*xx*'	The GPIO pins available for your program, labelled 'GP0' through to 'GP28'.
GP*xx*_ADC*x*	General-purpose input/output pin number '*xx*', with analogue input number '*x*'	A GPIO pin which ends in 'ADC' and a number can be used as an analogue input as well as a digital input or output – but not both at the same time.
ADC_VREF	Analogue-to-digital converter (ADC) voltage reference	A special input pin which sets a *reference voltage* for any analogue inputs.
AGND	Analogue-to-digital converter (ADC) 0 volts ground	A special ground connection for use with the ADC_VREF pin.
RUN	Enables or disables your Pico	The RUN header is used to start and stop your Pico from another microcontroller.

Several of the GPIO pins have additional functions, which you'll learn about later in the book. For a full pinout including these additional functions, see **Appendix B.**

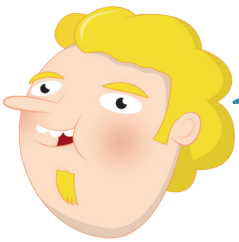
PIN GP0

Like counting in Python, your Pico's GPIO pins start at the number 0 rather than the number 1. Labelled on the underside of the board, they go from 0 to 29, although some of them aren't broken out as physical pins.



MISSING PINS

The general-purpose input/output pins on Pico are numbered based on the pins of the chip which powers it, an RP2040 microcontroller. Not all the pins available on RP2040 are brought out to your Pico's pins, however – which is why there's a gap in the numbering between the last basic general-purpose pin GP22 and the first analogue-capable pin GP26_ADC0.

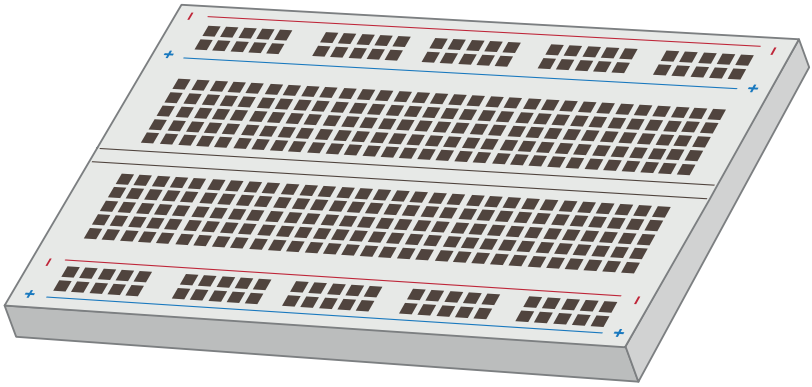


WARNING

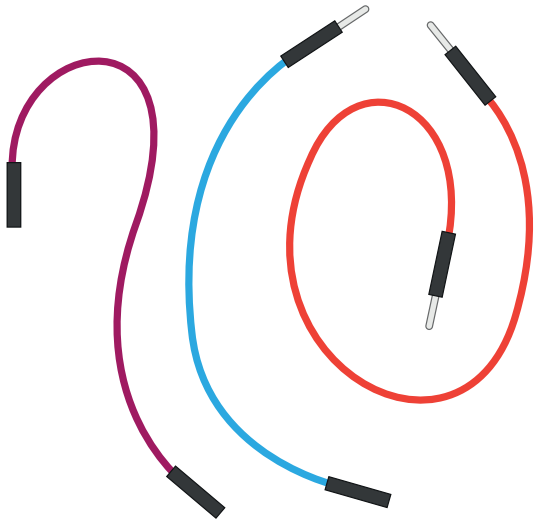
Your Pico's pins are designed to be a fun and safe way to experiment with physical computing, but should always be treated with care. Be careful not to bend the pins, especially when you're inserting your Pico into a breadboard. Never connect two pins directly together, accidentally or deliberately, unless you're told to do so in a project's instructions: this is known as a *short circuit* and, depending on the pins, can permanently damage your Pico.

Electronic components

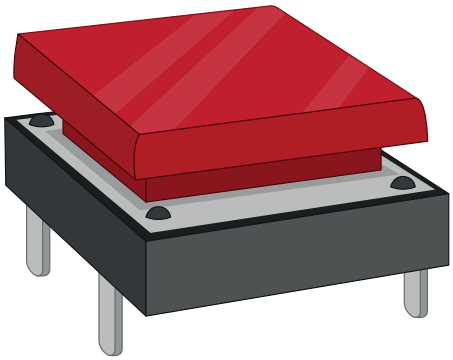
Your Pico is only part of what you'll need to begin working with physical computing; the other half is made up of electrical components, the devices you'll control from Pico's GPIO pins. There are thousands of different components available, but most physical computing projects are made using the following common parts.



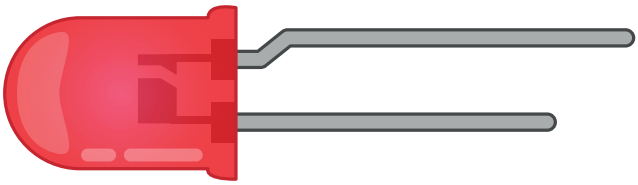
A *breadboard,* also known as a *solderless breadboard,* can make physical computing projects considerably easier. Rather than having a bunch of separate components which need to be connected with wires, a breadboard lets you insert components and have them connected through metal tracks which are hidden beneath its surface. Many breadboards also include sections for power distribution, making it easier to build your circuits. You don't need a breadboard to get started with physical computing, as you can use special wires to connect components directly to your Pico's GPIO pins, but it certainly makes things simpler and more stable.



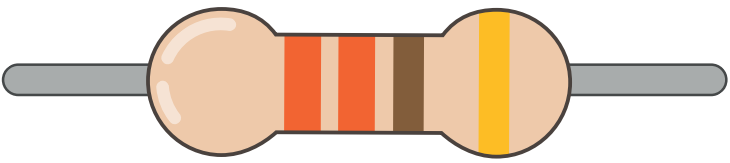
Jumper wires, also known as *jumper leads*, connect components to your Pico and, if you're not using a breadboard, to each other. They are available in three versions: male-to-female (M2F); female-to-female (F2F), which can be used to connect individual components to your Pico if you're not using a breadboard; and male-to-male (M2M), which is used to make connections from one part of a breadboard to another. Depending on your project, you may need all three types of jumper wire; if you're using a breadboard, you can usually get away with just M2F and M2M jumper wires.



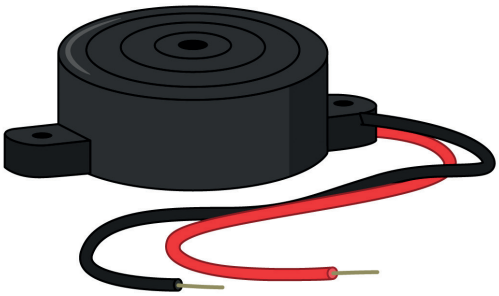
A *push-button switch*, also known as a *momentary switch,* is the type of switch you'd use to control a games console. Commonly available with two or four legs – either type will work with your Pico – the push-button is an input device: you can tell your program to watch out for it being pushed and then perform a task. Another common switch type is a latching switch; whereas a push-button is only active while you're holding it down, a *latching switch* – like you'd find in a light switch – activates when you toggle it once, then stays active until you toggle it again.



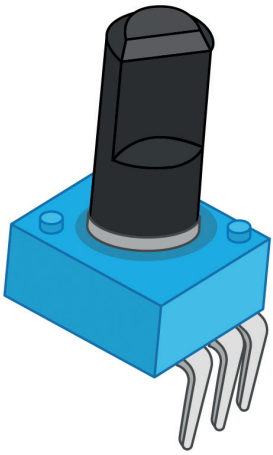
A *light-emitting diode (LED)* is an output device; you control it directly from your program. An LED lights up when it's on and you'll find them all over your house, ranging from the small ones which let you know when you've left your washing machine switched on to the large ones you might have lighting up your rooms. LEDs are available in a wide range of shapes, colours, and sizes, but not all are suitable for use with your Pico: avoid any which say they are designed for 5 V or 12 V power supplies.



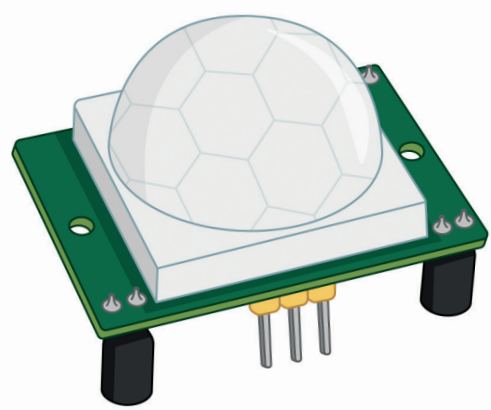
Resistors are components which control the flow of electrical current, and are available in different values measured using a unit called *ohms (Ω)*. The higher the number of ohms, the more resistance is provided. For Pico physical computing projects, their most common use is to prevent LEDs from drawing too much current and damaging themselves or your Pico; for this you'll want resistors rated at around 330 Ω, though many electrical suppliers sell handy packs containing a number of different commonly used values which give you more flexibility.



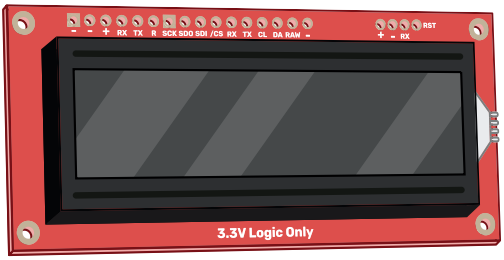
A *piezoelectric buzzer,* usually just called a *buzzer* or a *sounder,* is another output device. Whereas an LED produces light, though, a buzzer produces a noise – a buzzing noise, in fact. Inside the buzzer's plastic housing are a pair of metal plates; when switched on, these plates vibrate against each other to produce the buzzing sound. There are two types of buzzer: *active buzzers* and *passive buzzers.* Make sure to get an active buzzer, as these are the simplest to use.



A *potentiometer* is the sort of component you might find as a volume control on a music player, and can work as two different components. With two of its three legs connected, it acts as a *variable resistor* or *varistor,* a type of resistor which can be adjusted at any time by twisting the knob. With all three legs properly wired up, it becomes a *voltage divider* and outputs anything from 0 V to the full voltage input depending on the position of the knob.



A passive infrared sensor (PIR) is one of a variety of input devices known as *sensors*, designed to report on changes in whatever they are monitoring. In the case of a PIR sensor, it monitors movement: the sensor watches for changes in the area covered by its plastic lens, and sends a signal when it detects movement. PIR sensors are commonly found on burglar alarms, to find people moving in the dark.



An *I2C display* is a screen which talks to your Pico over a special communication system called the *inter-integrated circuit (I2C) bus*. This bus lets your Pico control the display panel, sending everything from writing to pictures for it to display. There are lots of types of display available, though a popular one – and the one found in this book – is the SerLCD from SparkFun, which has both I2C and SPI interfaces. Note that some displays only use the *serial peripheral interface (SPI)* bus rather than I2C; they'll still work with your Pico, but need the program to be changed.

Other common electrical components include motors, which need a special *driver* component before they can be connected to your Pico, current sensors which can detect how much power a circuit is using, inertial measurement units (IMUs) which track movement and orientation, and light-dependent resistors (LDRs) – input devices which operate like a reverse LED by detecting light rather than emitting it.

Sellers all over the world provide components for physical computing with Raspberry Pi, either as individual parts or in kits which provide everything you need to get started. To find sellers, visit **rpf.io/products**, click on Raspberry Pi 4, and you will get a list of Raspberry Pi partner online stores (approved resellers) for your country or region.

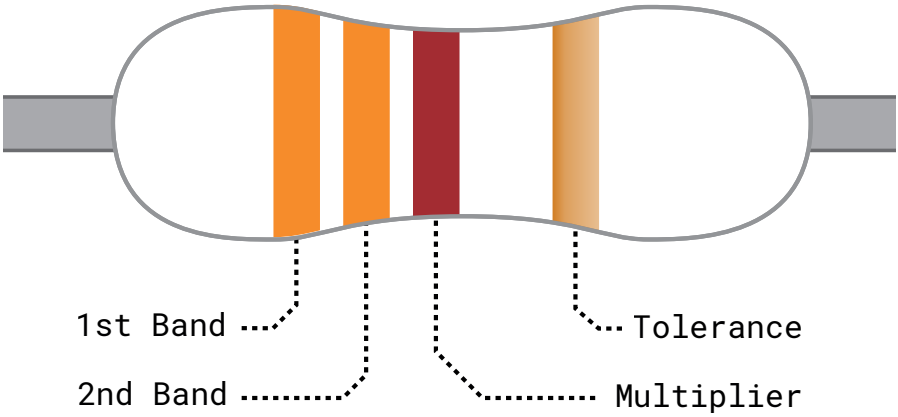
To complete the projects in this book, you should have at least:

- A Raspberry Pi Pico with male headers attached
- A micro USB cable
- A solderless breadboard
- A Raspberry Pi or other computer for programming
- Male-to-female (M2F) and male-to-male (M2M) jumper wires
- 3 × single-colour LEDs: red, green, and yellow or amber
- 1 × active piezoelectric buzzer
- 1 × 10 kΩ potentiometer, linear or logarithmic
- 3 × 330 Ω resistors
- At least one HC-SR501 PIR sensor
- 1 × SerLCD module
- WS2812B LEDs

You will also find it helpful to buy a cheap storage box with multiple compartments, so you can keep the components you're not using in your project safe and tidy. If you can, try to find one that will also fit the breadboard – that way you can tidy everything away each time you're done.

Reading resistor colour codes

Resistors come in a wide range of values, from zero-resistance versions which are effectively just pieces of wire to versions the size of your leg used in power stations. Very few of these resistors have their values printed on them in numbers, though: instead, they use a special code printed as coloured stripes or bands around the body of the resistor.



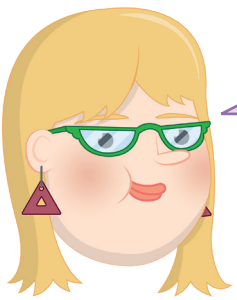
	1st/2nd Band	Multiplier	Tolerance
Black	0	$\times 10^{0}$	-
Brown	1	$\times 10^{1}$	±1%
Red	2	$\times 10^{2}$	±2%
Orange	3	$\times 10^{3}$	-
Yellow	4	$\times 10^{4}$	-
Green	5	$\times 10^{5}$	±0.5%
Blue	6	$\times 10^{6}$	±0.25%
Violet	7	$\times 10^{7}$	±0.1%
Grey	8	$\times 10^{8}$	±0.05%
White	9	$\times 10^{9}$	-
Gold	-	$\times 10^{-1}$	±5%
Silver	-	$\times 10^{-2}$	±10%
None	-	-	±20%

To read the value of a resistor, position it so the group of bands is to the left and the lone band is to the right. Starting from the first band, look its colour up in the '1st/2nd Band' column of the table to get the first and second digits. This example has two orange bands, which both mean a value of '3' for a total of '33'. If your resistor has four grouped bands instead of three, note down the value of the third band too – and for five- or six-band resistors see **rpf.io/5-6band.**

Moving onto the last grouped band – the third or fourth, depending on your resistor – look its colour up in the 'Multiplier' column. This tells you what number you need to multiply your current number by to get the actual value of the resistor. This example has a brown band, which means '$\times 10^1$'. That may look confusing, but it's simply *scientific notation:* '$\times 10^1$' simply means 'add one zero to the end of your number'. If it were blue, for $\times 10^6$', it would mean 'add six zeroes to the end of your number'.

33, from the orange bands, plus the added zero from the brown band gives us 330 – which is the value of the resistor, measured in ohms. The final band, on the right, is the *tolerance* of the resistor. This is simply how close to its rated value it is likely to be. Cheaper resistors might have a silver band, indicating it can be 10 percent higher or lower than its rating, or no last band at all, indicating it can be 20 percent higher or lower; the most expensive resistors have a grey band, indicating that it will be within 0.05 percent of its rating. For most hobbyist projects, accuracy isn't that important: any tolerance will usually work fine.

If your resistor value goes above 1000 ohms (1000 Ω), it is usually rated in kilohms (kΩ); if it goes above a million ohms, those are megohms (MΩ). A 2200 Ω resistor would be written as 2.2 kΩ; a 2200000Ω resistor would be written as 2.2 MΩ.



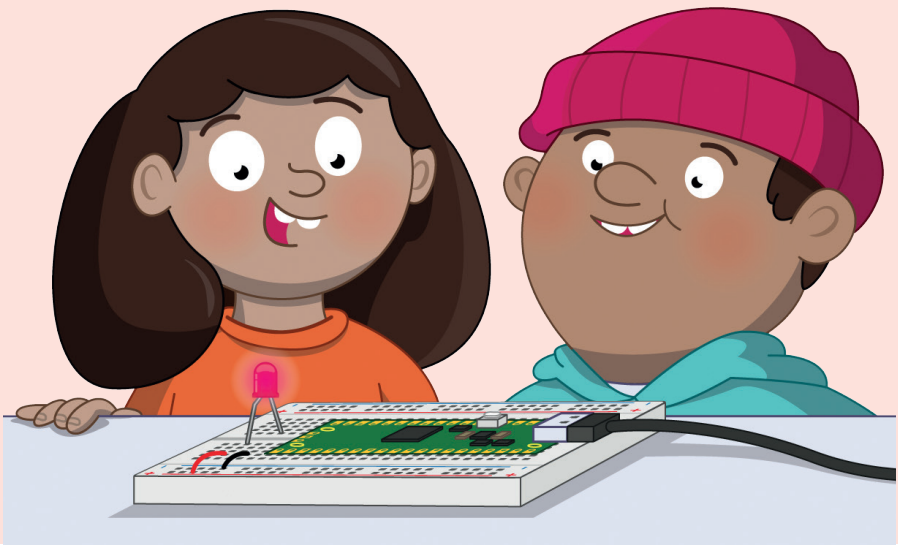
CAN YOU WORK IT OUT?

What colour bands would a 100 Ω resistor have? What colour bands would a 2.2 MΩ resistor have? If you wanted to find the cheapest resistors for your project, what colour tolerance band would you look for?

Chapter 4

Physical computing with Raspberry Pi Pico

Start connecting basic electronic components to your Raspberry Pi Pico and writing programs to control and sense them



Raspberry Pi Pico, with its RP2040 microcontroller, is designed with physical computing in mind. Its numerous general-purpose input/output (GPIO) pins let it talk to a range of components, allowing you to build up projects, from lighting LEDs to recording data about the world around you.

Physical computing is no more difficult to learn than traditional computing: if you could follow the examples in **Chapter 2**, you'll be able to build your own circuits and program them to do your bidding.

Your first physical computing program: Hello, LED!

Just as printing 'Hello, World' to the screen is a fantastic first step in learning a programming language, making an LED light up is the traditional introduction to learning physical computing. You can get started without any additional components, too: your Raspberry Pi Pico has a small LED, known as a *surface-mount device (SMD) LED*, on top.

Start by finding the LED: it's the small rectangular component to the left of the micro-USB port at the top of the board (**Figure 4-1**), marked with a label reading 'LED'. This small LED works just like any other: when it's powered on, it will glow; when it's powered off, it remains dark.

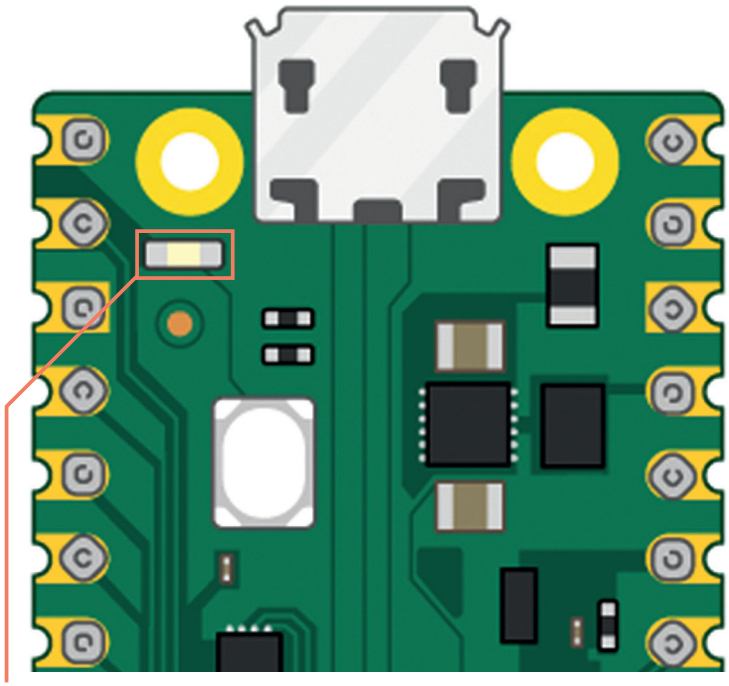


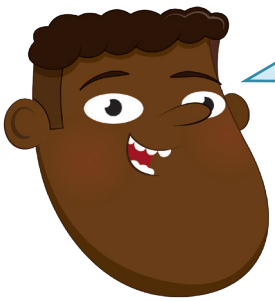
Figure 4-1: The on-board LED is found to the left of the micro-USB connector

The on-board LED is connected to one of RP2040's general-purpose input/output pins, GP25. This, you may recall from earlier chapters, is one of the 'missing' GPIO pins present on RP2040 but not broken out to a physical pin on the edge of your Pico. While you can't connect any external hardware to the pin, other than the on-board LED, it can be treated just the same as any other GPIO pin within your programs – and it makes a great way to add an output to your programs without needing any extra components.

Load Thonny and, if you haven't already done so, configure it to connect to your Pico as shown in **Chapter 2.** Click into the script area, and start your program with the following line:

```
import machine
```

This short line of code is key to working with MicroPython on your Pico: it loads, or *imports*, a collection of MicroPython code known as a *library* – in this case, the 'machine' library. The machine library contains all the instructions MicroPython needs to communicate with the Pico and other MicroPython-compatible devices, extending the language for physical computing. Without this line, you won't be able to control any of your Pico's GPIO pins – and you won't be able to make the on-board LED light up.



SELECTIVE IMPORTS

In both MicroPython and Python it's possible to import part of a library, rather than the whole library. Doing so can make your program use less memory, and allows you to mix and match functions from different libraries. The programs in this book import the whole library; elsewhere you may see programs which have lines like `from machine import Pin`; this tells MicroPython to import only the 'Pin' function from the 'machine' library, rather than the whole library.

The machine library exposes what is known as an *application programming interface (API)*. The name sounds complicated, but describes exactly what it does: it provides a way for your program, or the *application*, to communicate with the Pico via an *interface*.

The next line of your program provides an example of the machine library's API:

```
led_onboard = machine.Pin(25, machine.Pin.OUT)
```

This line defines an object called **`led_onboard`**, which offers a friendly name you can use to refer to the on-board LED later in your program. It's technically possible to use any name here – like **`susan, gupta`**, or **`fish_sandwich`** – but it's best to stick with names which describe the variable's purpose, to make the program easier to read and understand.

The second part of the line calls the Pin function in the machine library. This function, as its name suggests, is designed for handling your Pico's GPIO pins. At the moment, none of the GPIO pins – including GP25, the pin connected to the on-board LED – knows what they're supposed to be doing. The first argument, **`25`**, is the number of the pin you're setting up; the second, **`machine.Pin.OUT`**, tells Pico the pin should be used as an *output* rather than an *input*.

That line alone is enough to set the pin up, but it won't light the LED. To do that, you need to tell Pico to actually turn the pin on. Type the following code on the next line:

```
led_onboard.value(1)
```

It might not look it, but this line is also using the machine library's API. Your earlier line created the object **led_onboard** as an output on pin GP25; this line takes the object and sets its *value* to 1 for 'on' – it could also set the value to 0, for 'off'.

PIN NUMBERS

When talking about the GPIO pins on your Pico they're usually referred to using their full names: GP25 for the pin connected to the on-board LED, for example. In MicroPython, though, the letters G and P are dropped – so make sure you write '25' rather than 'GP25' in your program, or it won't work!

Click the Run button and save the program on your Pico as **Blink.py**. You'll see the LED light up. Congratulations – you've written your first physical computing program!

You'll notice, however, the LED stays lit: that's because your program tells Pico to turn it on, but never tells it to turn it off. You can add another line at the bottom of your program:

```
led_onboard.value(0)
```

Run the program this time, though, and the LED never seems to light up. That's because Pico works very, very quickly – far more quickly than you can see with the naked eye. The LED is lighting up, but for such a short time it appears to remain dark. To fix that, you need to slow your program down by introducing a *delay*.

Go back to the top of your program: click at the end of the first line and press **ENTER** to insert a new second line. On this line, type:

```
import utime
```

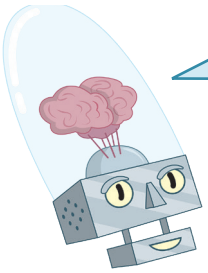
Like **import machine**, this line imports a new library into MicroPython: the 'utime' library. This library handles everything to do with time, from measuring it to inserting delays into your programs.

Go to the bottom of your program, and click on the end of the line **led_onboard.value(1)**, then press **ENTER** to insert a new line. Type:

```
utime.sleep(5)
```

This calls the **sleep** function from the utime library, which makes your program pause for whatever number of seconds you typed – in this case, five seconds.

Click the Run button again. This time you'll see the on-board LED on your Pico light up, stay lit for five seconds – try counting along – and go out again.



UTIME VS TIME

If you've programmed in Python before, you'll be used to using the 'time' library. The utime library is a version designed for microcontrollers like the Pico – the 'u' standing for 'µ', the Greek letter 'mu', which is used as a shorthand for 'micro'. If you forget and use `import time`, don't worry: MicroPython will automatically use the utime library instead.

Finally, it's time to make the LED blink. To do that, you'll need to create a loop. Rewrite your program so it matches the one below:

```
import machine
import utime

led_onboard = machine.Pin(25, machine.Pin.OUT)

while True:
    led_onboard.value(1)
    utime.sleep(5)
    led_onboard.value(0)
    utime.sleep(5)
```

Remember that the lines inside the loop need to be indented by four spaces, so MicroPython knows they form the loop. Click the Run icon again, and you'll see the LED switch on for five seconds, switch off for five seconds, and switch on again – constantly repeating in an infinite loop. The LED will continue to flash until you click the Stop icon to cancel your program and reset your Pico.

There's another way to handle the same job, too: using a *toggle*, rather than setting the LED's output to 0 or 1 explicitly. Delete the last four lines of your program and replace them so it looks like this:

```
import machine
import utime

led_onboard = machine.Pin(25, machine.Pin.OUT)

while True:
    led_onboard.toggle()
    utime.sleep(5)
```

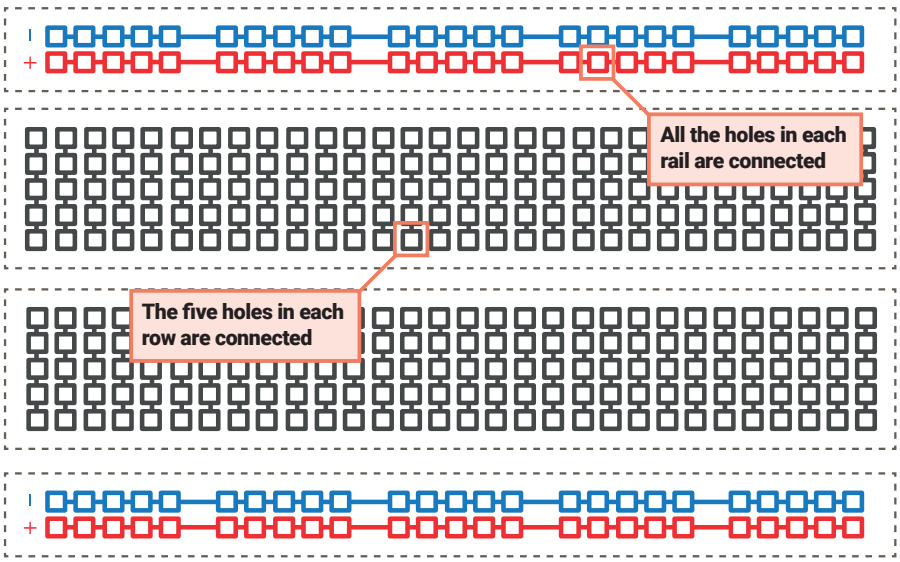
Run your program again. You'll see the same activity as before: the on-board LED will light up for five seconds, then go out for five seconds, then light up again in an infinite loop. This time, though, your program is two lines shorter: you've *optimised* it. Available on all digital output pins, **`toggle()`** simply switches between on and off: if the pin is currently on, **`toggle()`** switches it off; if it's off, **`toggle()`** switches it on.

CHALLENGE: LONGER LIGHT-UP

How would you change your program to make the LED stay on for longer? What about staying off for longer? What's the smallest delay you can use while still seeing the LED switch on and off?

Using a breadboard

The following projects in this chapter will be much easier to complete if you're using a breadboard to hold the components and make the electrical connections.



▲ **Figure 4-2:** The internal connections on a breadboard

A breadboard is covered with holes – spaced, to match components, 2.54 mm apart. Under these holes are metal strips which act like the jumper wires you've been using until now. These run in rows across the board, with most boards having a gap down the middle to split them in two halves.

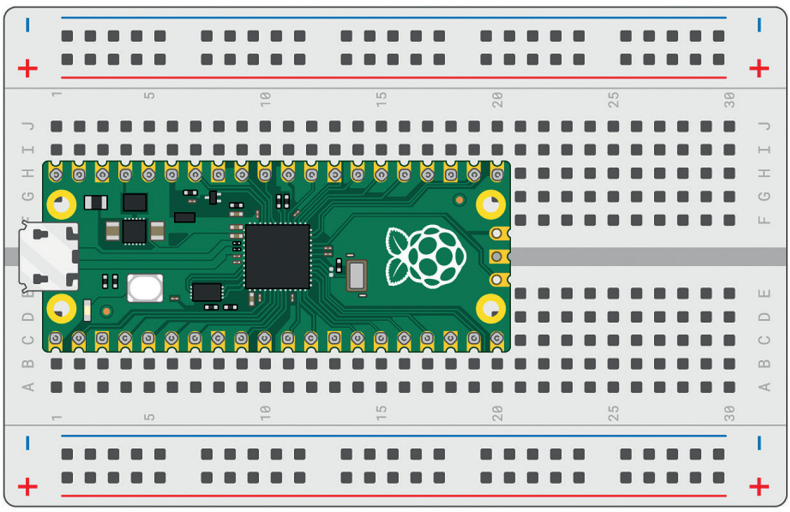
Many breadboards also have letters across the top and numbers down the sides. These allow you to find a particular hole: A1 is the top-left corner, B1 is the hole to the immediate right, while B2 is one hole down from there. A1 is connected to B1 by the hidden metal strips, but no hole marked with a 1 is ever connected to any hole marked with a 2 unless you add a jumper wire yourself.

Larger breadboards also have strips of holes down the sides, typically marked with red and black or red and blue stripes. These are the *power rails*, and are designed to make wiring easier: you can connect a single wire from one of your Pico's ground pins to one of the power rails – typically marked with a blue or black stripe and a minus symbol – to provide a common ground for lots of components on the breadboard, and you can do the same if your circuit needs 3.3 V or 5 V power. Note: All holes joined by a stripe are connected; a gap indicates a break.

Adding electronic components to a breadboard is simple: just line their leads (the sticky-out metal parts) up with the holes and gently push until the component is in place. For connections you need to make beyond those the breadboard makes for you, you can use male-to-male (M2M) jumper wires; for connections from the breadboard to external devices, like your Raspberry Pi Pico, use male-to-female (M2F) jumper wires.

Never try to cram more than one component lead or jumper wire into any single hole on the breadboard. Remember: holes are connected in rows, aside from the split in the middle, so a component lead in A1 is electrically connected to anything you add to B1, C1, D1, and E1.

Push your Pico into the breadboard so it straddles the middle gap and the micro USB port is at the very top of the board (see **Figure 4-3**). The top-left pin, Pin 0, should be in the breadboard row marked with a 1, if your breadboard is numbered. Before pushing your Pico down, make sure the header pins are all properly positioned – if you bend a pin, it can be difficult to straighten it again without it breaking.



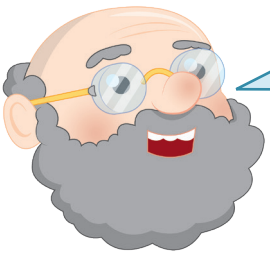
▲ **Figure 4-3:** Your Pico is designed to sit securely in a solderless breadboard

Gently push the Pico down until the plastic parts of the header pins are touching the breadboard. This means the metal parts of the header pins are fully inserted and making good electrical contact with the breadboard.

Next steps: an external LED

So far, you've been working with your Pico on its own – running MicroPython programs on its RP2040 microcontroller and toggling the on-board LED on and off. Microcontrollers are usually used with *external* components, though – and your Pico is no exception.

For this project, you'll need a breadboard, male-to-male (M2M) jumper wires, an LED, and a 330 Ω resistor – or as close to 330 Ω as you have available. If you don't have a breadboard, you can use female-to-female (F2F) jumper wires, but the circuit will be fragile and easy to break.



RESISTANCE IS VITAL

The resistor is a vital component in this circuit: it protects both your Pico and the LED by limiting the amount of electrical current the LED can draw. Without it, the LED can pull too much current and burn itself – or your Pico – out. When used like this, the resistor is known as a *current-limiting resistor*. The exact value of resistor you need depends on the LED you're using, but 330 Ω works for most common LEDs. The higher the value, the dimmer the LED; the lower the value, the brighter the LED.

Never connect an LED to your Pico without a current-limiting resistor, unless you know the LED has a built-in resistor of appropriate value.

Hold the LED in your fingers: you'll see one of its leads is longer than the other. The longer lead is known as the *anode*, and represents the positive side of the circuit; the shorter lead is the *cathode*, and represents the negative side. The anode needs to be connected to one of your Pico's GPIO pins via the resistor; the cathode needs to be connected to a ground pin.

Start by connecting the resistor: take one end (it doesn't matter which) and insert it into the breadboard in the same row as your Pico's GP15 pin at the bottom-left – if you're using a numbered breadboard with your Pico inserted at the very top, this should be row 20. Push the other end into a free row further down the breadboard – we're using row 24.

Take the LED, and push the longer leg – the anode – into the same row as the end of the resistor. Push the shorter leg – the cathode – into the same row but across the centre gap in the breadboard, so it's lined up but not electrically connected to the longer leg except through the LED itself. Finally, insert a male-to-male (M2M) jumper wire into the same row as the shorter leg of the LED, then either connect it directly to one of your Pico's ground pins (via another hole in its row) or to the negative side of your breadboard's power rail. If you connect it to the power rail, finish the circuit by connecting the rail to one of your Pico's ground pins. Your finished circuit should look like **Figure 4-4** (overleaf).

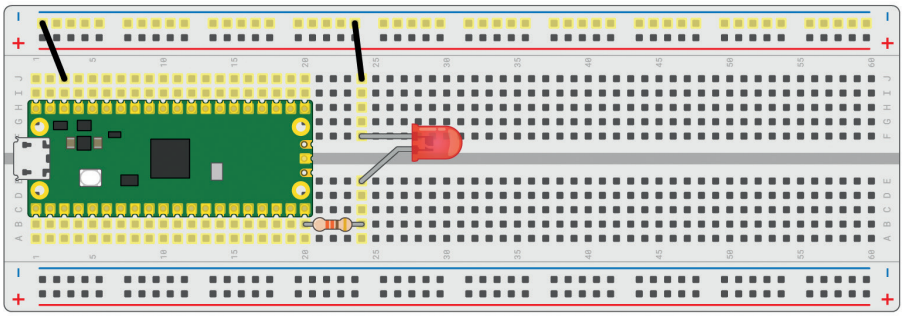


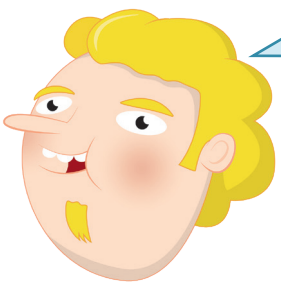
Figure 4-4: The finished circuit, with an LED and a resistor

Controlling an external LED in MicroPython is no different to controlling your Pico's internal LED: only the pin number changes. If you closed Thonny, reopen it and load your **Blink.py** program from earlier in the chapter. Find the line:

```
led_onboard = machine.Pin(25, machine.Pin.OUT)
```

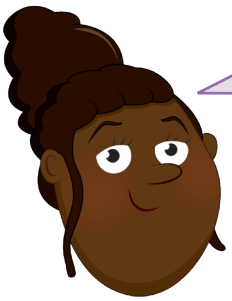
Edit the pin number, changing it from 25 – the pin connected to your Pico's internal LED – to 15, the pin to which you connected the external LED. Also edit the name you created: you're not using the on-board LED any more, so have it say `led_external` instead. You'll also have to change the name elsewhere in the program, until it looks like this:

```
import machine
import utime

led_external = machine.Pin(15, machine.Pin.OUT)

while True:
    led_external.toggle()
    utime.sleep(5)
```



NAMING CONVENTIONS

You don't really *need* to change the name in the program: it would run just the same if you'd left it at `led_onboard`, as it's only the pin number which truly matters. When you come back to the program later, though, it would be very confusing to have an object named `led_onboard` which lights up an external LED – so try to get into the habit of making sure your names match their purpose!



CHALLENGE: MULTIPLE LEDS

Can you modify the program to light up both the on-board and external LEDs at the same time? Can you write a program which lights up the on-board LED when the external LED is switched off, and vice versa? Can you extend the circuit to include more than one external LED? Remember, you'll need a current-limiting resistor for every LED you use!

Inputs: reading a button

Outputs like LEDs are one thing, but the 'input/output' part of 'GPIO' means you can use pins as inputs too. For this project, you'll need a breadboard, male-to-male jumper wires, and a push-button switch. If you don't have a breadboard, you can use female-to-female (F2F) jumper wires, but the button will be much harder to press without accidentally breaking the circuit.

Remove any other components from your breadboard except your Pico, and begin by adding the push-button switch. If your push-button has only two legs, make sure they're in different-numbered rows on the breadboard somewhere below your Pico. If it has four legs, turn it so the sides the legs come from are along the breadboard's rows and the flat leg-free sides are at the top and bottom before pushing it home so it straddles the centre divide of the breadboard.

Connect the positive power rail of your breadboard to your Pico's 3V3 pin, and from there to one of the legs of the switch; then connect the other leg to pin GP14 on your Pico – it's the one just above the pin you used for the LED project, and should be in row 19 of your breadboard.

If you're using push-button with four legs, your circuit will only work if you use the correct pair of legs: the legs are connected in pairs, so you need to either use the two legs on the same side or (as seen in **Figure 4-5**) diagonally opposite legs.

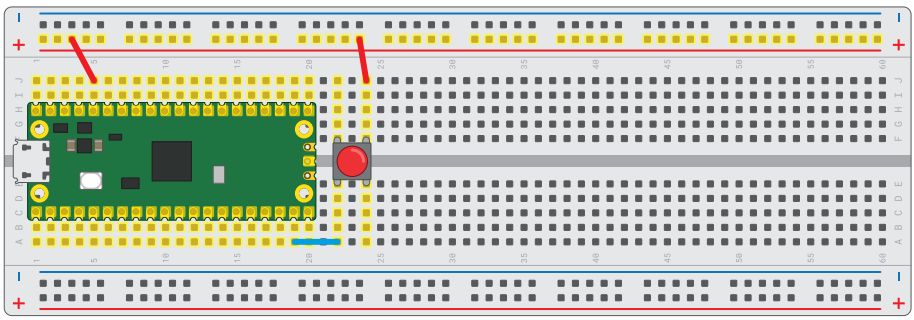


Figure 4-5: Wiring a four-leg push-button switch to GP14

RESISTANCE IS HIDDEN

Unlike an LED, a push-button switch doesn't need a current-limiting resistor. It does still need a resistor, though: it needs what is known as a *pull-up* or *pull-down* resistor, depending on how your circuit works. Without a pull-up or pull-down resistor, an input is known as *floating* – which means it has a 'noisy' signal which can trigger even when you're not pushing the button.

So where's the resistor in this circuit? Hidden in your Pico. Just like it has an on-board LED, your Pico includes an on-board *programmable resistor* connected to each GPIO pin. These can be set in MicroPython to pull-down resistors or pull-up resistors as required by your circuit.

What's the difference? A pull-down resistor connects the pin to ground, meaning when the push-button isn't pressed, the input will be 0. A pull-up resistor connects the pin to 3V3, meaning when the push-button isn't pressed, the input will be 1.

All the circuits in this book use the programmable resistors in the pull-down mode.

Load Thonny, if you haven't already, and start a new program with the usual line:

```
import machine
```

Next, you need to use the machine API to set up a pin as an input, rather than an output:

```
button = machine.Pin(14, machine.Pin.IN, machine.Pin.PULL_DOWN)
```

This works in the same way as your LED projects: an object called 'button' is created, which includes the pin number – GP14, in this case – and configures it as an input with the resistor set to pull-down. Creating the object, though, doesn't mean it will do anything by itself – just as creating the LED objects earlier didn't make the LEDs light up.

To actually read the button, you need to use the machine API again – this time using the **value** function to read, rather than set, the value of the pin. Type the following line:

```
print(button.value())
```

Click the Run icon and save your program as **Button.py** – remembering to make sure it saves on the MicroPython device, your Pico. Your program will print out a single number: the value of the

input on GP14. Because the input is using a pull-down resistor, this value will be 0 – letting you know the button isn't pushed.

Hold down the button with your finger, and press the Run icon again. This time, you'll see the value 1 printed to the Shell: pushing the button has completed the circuit and changed the value read from the pin.

To read the button continuously, you'll need to add a loop to your program. Edit the program so it reads as below:

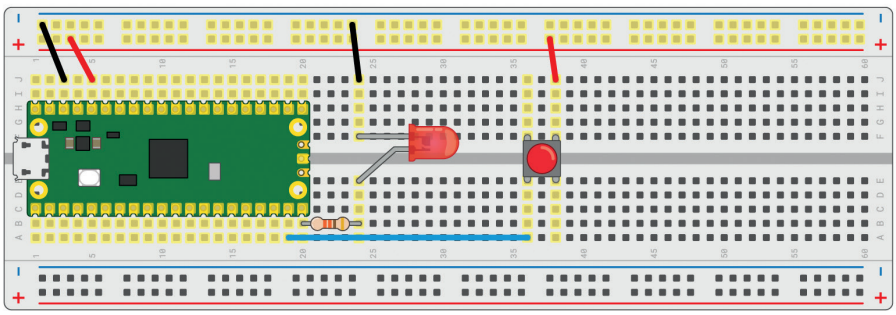
```
import machine
import utime

button = machine.Pin(14, machine.Pin.IN, machine.Pin.PULL_DOWN)

while True:
    if button.value() == 1:
        print("You pressed the button!")
        utime.sleep(2)
```

Click the Run button again. This time, nothing will happen until you press the button; when you do, you'll see a message printed to the Shell area. The delay, meanwhile, is important: remember, your Pico runs a lot faster than you can read, and without the delay even a brief press of the button can print hundreds of messages to the Shell!

You'll see the message print every time you press the button. If you hold the button down for longer than the two-second delay, it will print the message every two seconds until you let go of the button.

Inputs and outputs: putting it all together

Most circuits have more than one component, which is why your Pico has so many GPIO pins. It's time to put everything you've learned together to build a more complex circuit: a device which switches an LED on and off with a button.



▲ **Figure 4-6:** The finished circuit, with both a button and an LED

In effect, this circuit combines both of the previous circuits into one. You may remember you used pin GP15 to drive the external LED, and pin GP14 to read the button; now rebuild your circuit so both the LED and the button are on the breadboard at the same time, still connected to GP15 and GP14 (see **Figure 4-6**). Don't forget the current-limiting resistor for the LED!

Start a new program in Thonny, and begin importing the two libraries your program will need:

```
import machine
import utime
```

Next, set up both the input and output pins:

```
led_external = machine.Pin(15, machine.Pin.OUT)
button = machine.Pin(14, machine.Pin.IN, machine.Pin.PULL_DOWN)
```

Then create a loop which reads the button:

```
while True:
    if button.value() == 1:
```

Rather than printing a message to the Shell, though, this time you're going to toggle the output pin and the LED connected to it based on the value of the input pin. Type the following, remembering it will need to be indented by eight spaces – which Thonny should have automatically handled when you pressed **ENTER** at the end of the line above:

```
        led_external.value(1)
        utime.sleep(2)
```

That's enough to turn the LED on, but you'll also need to turn it off again when the button isn't being pressed. Add the following new line, using the **BACKSPACE** key to delete four of the eight spaces – meaning the line will not be part of the if statement, but will form part of the infinite loop:

```
    led_external.value(0)
```

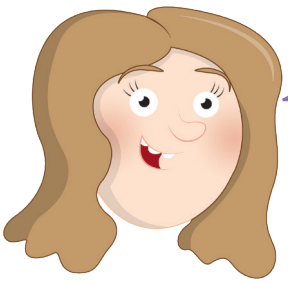
Your finished program should look like this:

```
import machine
import utime

led_external = machine.Pin(15, machine.Pin.OUT)
button = machine.Pin(14, machine.Pin.IN, machine.Pin.PULL_DOWN)
```

```
while True:
    if button.value() == 1:
        led_external.value(1)
        utime.sleep(2)
    led_external.value(0)
```

Click the Run icon and save the program as **Switch.py** on your Pico. At first, nothing will happen; push the button, though, and you'll see the LED light up. Let go of the button; after two seconds, the LED will go out again until you press the button again.

Congratulations: you've built your first circuit which controls one pin based on the input from another – a building block for bigger things!



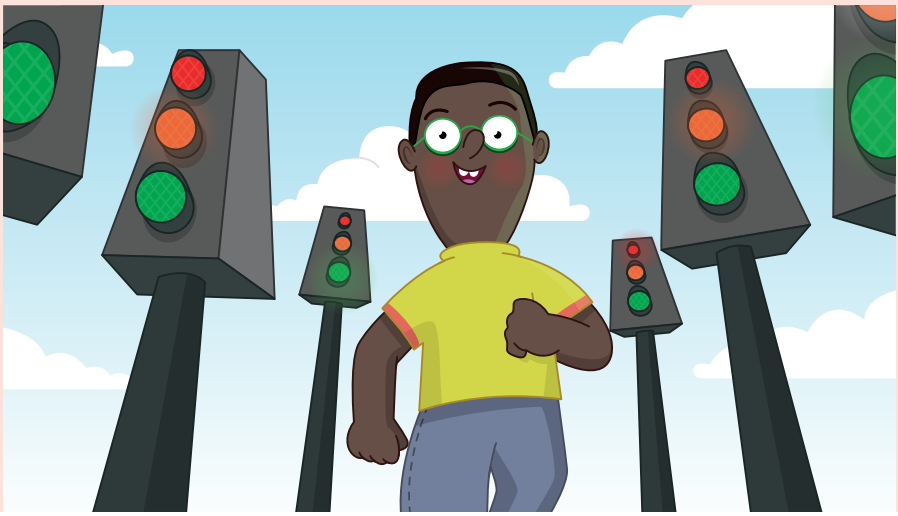
CHALLENGE: BUILDING IT UP

Can you modify your program so it both lights the LED and prints a status message to the Shell? What would you need to change to make the LED stay on when the button isn't pressed and switch off when it is? Can you add more buttons and LEDs to the circuit?

Chapter 5

Traffic light controller

Create your own mini pedestrian crossing system using multiple LEDs and a push-button



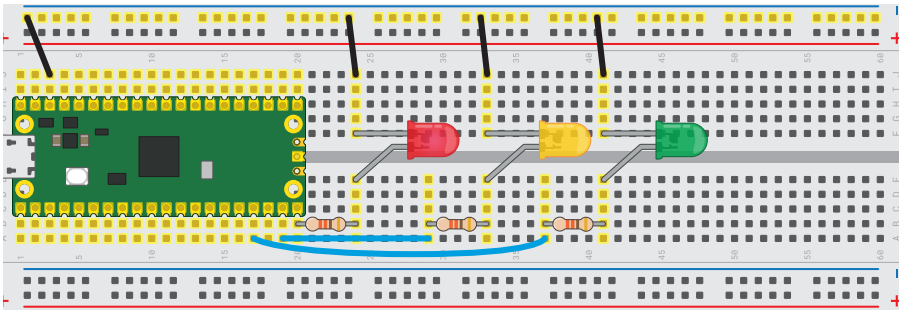
Microcontrollers can be found in almost all the electronic items you use on a daily basis – including traffic lights. A traffic light controller is a specially built system which changes the lights on a timer, watches for pedestrians looking to cross, and can even adjust the timing of the lights depending on how much traffic there is – talking to nearby traffic light systems to ensure the whole traffic network keeps flowing smoothly.

While building a large-scale traffic management system is a pretty advanced project, it's simplicity itself to build a miniature simulator powered by your Raspberry Pi Pico. With this project, you'll see how to control multiple LEDs, set different timings, and how to monitor a push-button input while the rest of the program continues to run using a technique known as *threading*.

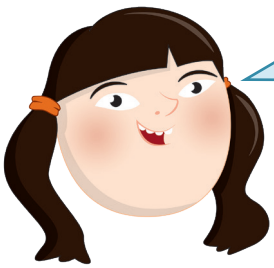
For this project, you'll need your Pico; a breadboard; a red, a yellow or amber, and a green LED; three 330 Ω resistors; an active piezoelectric buzzer; and a selection of male-to-male (M2M) jumper wires. You'll also need a micro USB cable, and to connect your Pico to your Raspberry Pi or other computer running the Thonny MicroPython IDE.

A simple traffic light

Start by building a simple traffic light system, as shown in **Figure 5-1**. Take your red LED and insert it into the breadboard so it straddles the centre divide. Use one of the 330 Ω resistors, and a jumper wire if you need to make a longer connection, to connect the longer leg – the anode – of the LED to the pin at the bottom-left of your Pico as seen from the top with the micro USB cable uppermost, GP15. If you're using a numbered breadboard and have your Pico inserted at the very top, this will be breadboard row 20.



▲ **Figure 5-1:** A basic three-light traffic light system



WARNING

Always remember that an LED needs a current-limiting resistor before it can be connected to your Pico. If you connect an LED without a current-limiting resistor in place, the best outcome is the LED will burn out and no longer work; the worst outcome is it could do the same to your Pico.

Take a jumper wire and connect the shorter leg – the cathode – of the red LED to your breadboard's ground rail. Take another, and connect the ground rail to one of your Pico's ground (GND) pins – in **Figure 5-1**, we've used the ground pin on row three of the breadboard.

You've now got one LED connected to your Pico, but a real traffic light has at least two more for a total of three: a red light to tell the traffic to stop, an amber or yellow light to tell the traffic the light is about to change, and a green LED to tell the traffic it can go again.

Take your amber or yellow LED and wire it to your Pico in the same way as the red LED, making sure the shorter leg is the one connecting to the ground rail of the breadboard and that you've got the 330 Ω resistor in place to protect it. This time, though, wire the longer leg – via the resistor – to the pin next to the one to which you wired the red LED, GP14.

Finally, take the green LED and wire it up the same way again – remembering the 330 Ω resistor – to pin GP13. This isn't the pin right next to pin GP14, though – that pin is a ground (GND) pin, which you can see if you look closely at your Pico: the ground pins all have a square shape to their pads, while the other pins are round.

When you've finished, your circuit should match **Figure 5-1**: a red, a yellow or amber, and a green LED, all wired to different GPIO pins on your Pico via individual 330 Ω resistors and connected to a shared ground pin via your breadboard's ground rail.

To program your traffic lights, connect your Pico to your Raspberry Pi or other computer and load Thonny. Create a new program, and start by importing the machine library so you can control your Pico's GPIO pins:

```
import machine
```

You'll also need to import the utime library, so you can add delays between the lights going on and off:

```
import utime
```

As with any program using your Pico's GPIO pins, you'll need to set each pin up before you can control it:

```
led_red = machine.Pin(15, machine.Pin.OUT)
led_amber = machine.Pin(14, machine.Pin.OUT)
led_green = machine.Pin(13, machine.Pin.OUT)
```

These lines set pins GP15, GP14, and GP13 up as outputs, and each is given a descriptive name to make it easier to read the code: 'led', so you know the pins control an LED, and then the colour of the LED.

Real traffic lights don't run through once and stop – they keep going, even when there's no traffic there and everyone's asleep. So that your program does the same, you'll need to set up an infinite loop:

```
while True:
```

Each of the lines beneath this need to be indented by four spaces, so MicroPython knows they form part of the loop; when you press the **ENTER** key Thonny will automatically indent the lines for you.

```
    led_red.value(1)
    utime.sleep(5)
    led_amber.value(1)
    utime.sleep(2)
    led_red.value(0)
    led_amber.value(0)
    led_green.value(1)
```

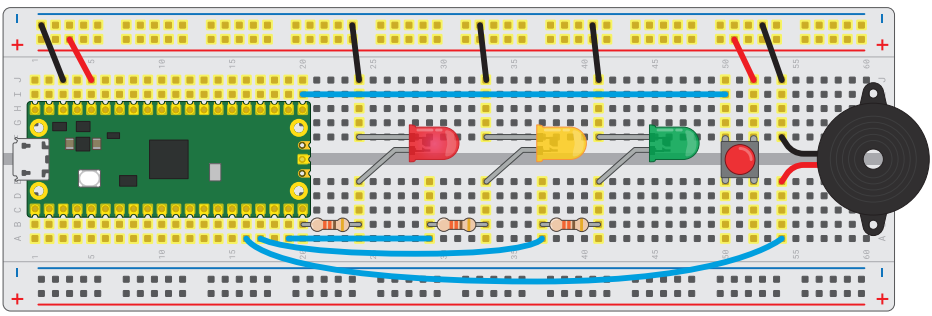
```
utime.sleep(5)
led_green.value(0)
led_amber.value(1)
utime.sleep(5)
led_amber.value(0)
```

Click the Run icon and save your program to your Pico as **Traffic_Lights.py.** Watch the LEDs: first the red LED will light up, telling the traffic to stop; next, the amber LED will come on to warn drivers the lights are about to change; next both LEDs switch off and the green LED comes on to let traffic know it can pass; then the green LED goes off and the amber one comes on to warn drivers the lights are about to change again; finally, the amber LED goes off – and the loop restarts from the beginning, with the red LED coming on.

The pattern will loop until you press the Stop button, because it forms an infinite loop. It's based on the traffic light pattern used in real-world traffic control systems in the UK and Ireland, but sped up – giving cars just five seconds to pass through the lights wouldn't let the traffic flow very freely!

Real traffic lights aren't just there for road vehicles, though: they are also there to protect pedestrians, giving them an opportunity to cross a busy road safely. In the UK, the most common type of these lights are known as *pedestrian-operated user-friendly intelligent crossings* or *puffin crossings*.

To turn your traffic lights into a puffin crossing, you'll need two things: a push-button switch, so the pedestrian can ask the lights to let them cross the road; and a buzzer, so the pedestrian knows when it's their turn to cross. Wire those into your breadboard as in **Figure 5-2**, with the switch wired to pin GP16 and the 3V3 rail of your breadboard, and the buzzer wired to pin GP12 and the ground rail of your breadboard.



▲ **Figure 5-2:** A puffin crossing traffic light system

If you run your program again, you'll find the button and buzzer do nothing. That's because you haven't yet told your program how to use them. In Thonny, go back to the lines where you set up your LEDs and add the following two new lines below:

```
button = machine.Pin(16, machine.Pin.IN, machine.Pin.PULL_DOWN)
buzzer = machine.Pin(12, machine.Pin.OUT)
```

This sets the button on pin GP16 up as an input, and the buzzer on pin GP12 as an output. Remember, your Raspberry Pi Pico has built-in programmable resistors for its inputs, which we are setting to pull-down mode for the projects in this book. This means that the pin's voltage is pulled down to 0 V (and its logic level is 0), unless it is connected to 3.3V power (in which case its logic level will be 1 until disconnected).

Next, you need a way for your program to constantly monitor the value of the button. Previously, all your programs have worked step-by-step through a list of instructions – only ever doing one thing at a time. Your traffic light program is no different: as it runs, MicroPython walks through your instructions step-by-step, turning the LEDs on and off.

For a basic set of traffic lights, that's enough; for a puffin crossing, though, your program needs to be able to record whether the button has been pressed in a way that doesn't interrupt the traffic lights. To make that work, you'll need a new library: _thread. Go back to the section of your program where you import the machine and utime libraries, and import the _thread library:

```
import _thread
```

A *thread* or *thread of execution* is, effectively, a small and partially independent program. You can think of the loop you wrote earlier, which controls the lights, as the *main thread* of your program – and using the _thread library you can create an additional thread, running at the same time.

An easy way to visualise threads is to think of each one as a separate worker in a kitchen: while the chef is preparing the main dish, someone else is working on a sauce. At the moment, your program has only one thread – the one which controls the traffic lights. The RP2040 microcontroller which powers your Pico, however, has two processing cores – meaning, like the chef and the sous chef in the kitchen, you can run two threads at the same time to get more work done.

Before you can make another thread, you'll need a way for the new thread to pass information back to the main thread – and you can do this using *global variables*. The variables you've been working with prior to this are known as *local variables*, and only work in one section of your program; a global variable works everywhere, meaning one thread can change the value and another can check to see if it has been changed.

To start, you need to create a global variable. Below your **`buzzer =`** line, add the following:

```
global button_pressed
button_pressed = False
```

This sets up **`button_pressed`** as a global variable, and gives it a default value of **`False`** – meaning when the program starts, the button hasn't yet been pushed. The next step is to define

your thread, by adding the following lines directly below – adding a blank line, if you want, to make your program more readable:

```
def button_reader_thread():
    global button_pressed
    while True:
        if button.value() == 1:
            button_pressed = True
```

The first line you've added defines your thread and gives it a name which describes its purpose: a thread to read the button input. Like when writing a loop, MicroPython needs everything contained within the thread to be indented by four spaces – so it knows where the thread begins and ends.

The next line lets MicroPython know you're going to be changing the value of the global **`button_pressed`** variable. If you only want to check the value, you wouldn't need this line – but without it you can't make any changes to the variable.

Next, you've set up a new loop – which means a new four-space indent needs to follow, for eight in total, so MicroPython knows both that the loop is part of the thread and the code below is part of the loop. This nesting of code in multiple levels of indentation is very common in MicroPython, and Thonny will do its best to help you by automatically adding a new level each time it's needed – but it's up to you to remember to delete the spaces it adds when you're finished with a particular section of the program.

The next line is a conditional which checks to see if the value of the button is 1. Because your Pico is using an internal pull-down resistor, when the button isn't being pressed the value read is 0 – meaning the code under the conditional never runs. Only when the button is pressed will the final line of your thread run: a line which sets the **`button_pressed`** variable to **`True`**, letting the rest of your program know the button has been pushed.

You might notice there's nothing in the thread to reset the **`button_pressed`** variable back to **`False`** when the button is released after being pushed. There's a reason for that: while you can push the button of a puffin crossing at any time during the traffic light cycle, it only takes effect when the light has gone red and it's safe for you to cross. All your new thread needs to do is to change the variable when the button has been pushed; your main thread will handle resetting it back to **`False`** when the pedestrian has safely crossed the road.

Defining a thread doesn't set it running: it's possible to start a thread at any point in your program, and you'll need to specifically tell the _thread library when you want to launch the thread. Unlike running a normal line of code, running the thread doesn't stop the rest of the program: when the thread starts, MicroPython will carry on and run the next line of your program even as it runs the first line of your new thread.

Create a new line below your thread, deleting all of the indentation Thonny has automatically added for you, which reads:

```
_thread.start_new_thread(button_reader_thread, ())
```

This tells the _thread library to start the thread you defined earlier. At this point, the thread will start to run and quickly enter its loop – checking the button thousands of times a second to see if it's been pressed yet. The main thread, meanwhile, will carry on with the main part of your program.

Click the Run button now. You'll see the traffic lights carry on their pattern exactly as before, with no delay or pauses. If you press the button, though, nothing will happen – because you haven't added the code to actually react to the button yet.

Go to the start of your main loop, directly underneath the line **`while True:`**, and add the following code – remembering to pay attention to the nested indentation, and deleting the indentation Thonny has added when it's no longer required:

```
    if button_pressed == True:
        led_red.value(1)
        for i in range(10):
            buzzer.value(1)
            utime.sleep(0.2)
            buzzer.value(0)
            utime.sleep(0.2)
        global button_pressed
        button_pressed = False
```

This chunk of code checks the **`button_pressed`** global variable to see if the push-button switch has been pressed at any time since the loop last ran. If it has, as reported by the button reading thread you made earlier, it begins running a section of code which starts by turning the red LED on to stop traffic and then beeps the buzzer ten times – letting the pedestrian know it's time to cross.

Finally, the last two lines reset the **`button_pressed`** variable back to **`False`** – so the next time the loop runs it won't trigger the pedestrian crossing code unless the button has been pushed again. You'll see you didn't need the line **`global button_pressed`** to check the status of the variable in the conditional; it's only needed when you want to change the variable and have that change affect other parts of your program.

Your finished program should look like this:

```
import machine
import utime
import _thread

led_red = machine.Pin(15, machine.Pin.OUT)
led_amber = machine.Pin(14, machine.Pin.OUT)
```

```
led_green = machine.Pin(13, machine.Pin.OUT)
button = machine.Pin(16, machine.Pin.IN, machine.Pin.PULL_DOWN)
buzzer = machine.Pin(12, machine.Pin.OUT)

global button_pressed
button_pressed = False

def button_reader_thread():
    global button_pressed
    while True:
        if button.value() == 1:
            button_pressed = True

_thread.start_new_thread(button_reader_thread, ())

while True:
    if button_pressed == True:
        led_red.value(1)
        for i in range(10):
            buzzer.value(1)
            utime.sleep(0.2)
            buzzer.value(0)
            utime.sleep(0.2)
        global button_pressed
        button_pressed = False
    led_red.value(1)
    utime.sleep(5)
    led_amber.value(1)
    utime.sleep(2)
    led_red.value(0)
    led_amber.value(0)
    led_green.value(1)
    utime.sleep(5)
    led_green.value(0)
    led_amber.value(1)
    utime.sleep(5)
    led_amber.value(0)
```

Click the Run icon. At first, the program will run as normal: the traffic lights will go on and off in the usual pattern. Press the push-button switch: if the program is currently in the middle of its loop, nothing will happen until it reaches the end and loops back around again – at

which point the light will go red and the buzzer will beep to let you know it's safe to cross the road.

The conditional section of code for crossing the road runs before the code you wrote earlier for turning the lights on and off in a cyclic pattern: after it's finished, the pattern will begin as usual with the red LED staying lit for a further five seconds on top of the time it was lit while the buzzer was going. This mimics how a real puffin crossing works: the red light remains lit even after the buzzer has stopped sounding, so anyone who started to cross the road while the buzzer was going has time to reach the other side before the traffic is allowed to go.

Let the traffic lights loop through their cycle a few more times, then press the button again to trigger another crossing. Congratulations: you've built your own puffin crossing!

CHALLENGE: CAN YOU IMPROVE IT?

Can you change the program to give the pedestrian longer to cross? Can you find information about other countries' traffic light patterns and reprogram your lights to match? Can you add a second button, so the pedestrian on the other side of the road can signal they want to cross too?

Chapter 6
Reaction game

Build a simple reaction timing game using an LED and push-buttons, for one or two players

Microcontrollers aren't only found in industrial devices: they power plenty of electronics around the home, including toys and games. In this chapter you're going to build a simple reaction timing game, seeing who among your friends will be the first to press a button when a light goes off.

The study of reaction time is known as *mental chronometry* and while it forms a hard science, it is also the basis of plenty of skill-based games – including the one you're about to build. Your reaction time – the time it takes your brain to process the need to do something and send the signals to make that something happen – is measured in milliseconds: the average human reaction time is around 200–250 milliseconds, though some people enjoy considerably faster reaction times that will give them a real edge in the game!

For this project you'll need your Pico; a breadboard; an LED of any colour; a single 330 Ω resistor; two push-button switches; and a selection of male-to-male (M2M) jumper wires. You'll also need a micro USB cable, and to connect your Pico to your Raspberry Pi or other computer running the Thonny MicroPython IDE.

A single-player game

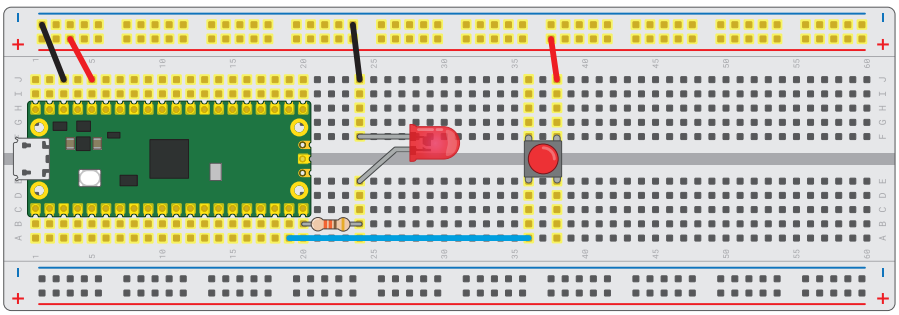
Start by placing your LED into your breadboard so that it straddles the centre divide. Remember that LEDs only work when they're the right way around: make sure you know which is the longer leg, or the anode, and which is the shorter leg, the cathode.

Using a 330 Ω current-limiting resistor, to protect both the LED and your Pico, wire the longer leg of the LED to pin GP15 at the bottom-left of your Pico as seen from the top with the micro USB cable uppermost. If you're using a numbered breadboard and have your Pico inserted at the very top, this will be breadboard row 20.

WARNING

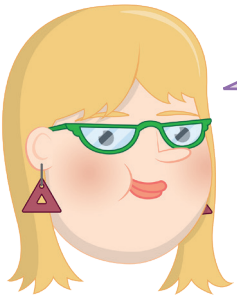
It bears repeating that an LED always needs a current-limiting resistor before it can be connected to your Pico. Without the resistor, the LED could burn out – or your Pico could be damaged.

Take a jumper wire and connect the shorter leg of the LED to your breadboard's ground rail. Take another, and connect the ground rail to one of your Pico's ground (GND) pins – in **Figure 6-1**, we've used the ground pin on row three of the breadboard.



▲ **Figure 6-1:** A single-player reaction game

Next, add the push-button switch as shown in **Figure 6-1**. Take a jumper wire and connect one of the push-button's switches to pin GP14, right next to the pin you used for your LED. Use another jumper wire to connect the other leg – the one diagonally opposite the first, if you're using a four-leg push-button switch – to your breadboard's power rail. Finally, take a last jumper wire and connect the power rail to your Pico's 3V3 pin.



WHY 3V3?

Remember that switches, like LEDs, need resistors to operate correctly, and that your Pico has programmable resistors on all its GPIO pins. In the projects for this book, we are setting them to pull-down resistors, meaning the pin has to be pulled high when the push-button switch is pressed – which is what wiring the switch to the 3V3 pin via the breadboard's power rail does.

Your circuit now has everything it needs to act as a simple single-player game: the LED is the output device, taking the place of the TV you would normally use with a games console; the push-button switch is the controller; and your Pico is the games console, albeit one considerably smaller than you'd usually see!

Now you need to actually write the game. As always, connect your Pico to your Raspberry Pi or other computer and load Thonny. Create a new program, and start it by importing the machine library so you can control your Pico's GPIO pins:

```
import machine
```

You're also going to need the utime library:

```
import utime
```

In addition, you'll need a new library: urandom, which handles creating random numbers – a key part of making a game fun, and used in this game to prevent a player who has played it before from simply counting down a fixed number of seconds from clicking the Run button.

Next, set up the two pins you're using: GP15 for the LED, and GP14 for the push-button switch.

```
led = machine.Pin(15, machine.Pin.OUT)
button = machine.Pin(14, machine.Pin.IN, machine.Pin.PULL_DOWN)
```

In previous chapters, you've handled push-button switches in either the main program or in a separate thread. This time, though, you're going to take a different and more flexible approach: *interrupt requests*, or *IRQs*.

The name sounds complex, but it's really simple: imagine you're reading a book, page by page, and someone comes up to you and asks you a question. That person is performing an interrupt request: asking you to stop what you're doing, answer their question, then letting you go back to reading your book.

A MicroPython interrupt request works in exactly the same way: it allows something, in this case the press of a push-button switch, to interrupt the main program. In some ways it's similar to a thread, in that there's a chunk of code which sits outside the main program. Unlike a thread, though, the code isn't constantly running: it only runs when the interrupt is triggered.

Start by defining a *handler* for the interrupt. This, known as a *callback function*, is the code which runs when the interrupt is triggered. As with any kind of nested code, the handler's code – everything after the first line – needs to be indented by four spaces; Thonny will do this for you automatically.

```
def button_handler(pin):
    button.irq(handler=None)
    print(pin)
```

This simple two-line handler starts by turning the interrupt off, so that it only triggers once, and then prints out information about the pin responsible for triggering the interrupt. That's not too important at the moment – you only have one pin configured as an input, GP14, so the interrupt will always come from that pin – but lets you test your interrupt easily.

Continue your program below, remembering to delete the four-line indent that Thonny has automatically created – the following code is not part of the handler:

```
led.value(1)
utime.sleep(urandom.uniform(5, 10))
led.value(0)
```

This code will be immediately familiar to you: the first line turns the LED, connected to pin GP15, on; the next line pauses the program; the last line turns the LED off again – the player's signal to push the button. Rather than using a fixed delay, however, it makes use of the urandom library to pause the program for between five and ten seconds – the 'uniform' part referring to a uniform distribution between those two numbers.

At the moment, though, there's nothing watching for the button being pushed. You need to set up the interrupt for that, by typing in the following line at the bottom of your program:

```
button.irq(trigger=machine.Pin.IRQ_RISING, handler=button_handler)
```

Setting up an interrupt requires two things: a *trigger* and a *handler*. The trigger tells your Pico what it should be looking for as a valid signal to interrupt what it's doing; the handler, which you defined earlier in your program, is the code which runs after the interrupt is triggered.
In this program your trigger is **`IRQ_RISING:`** this means that the interrupt is triggered when the pin's value rises from low, its default state thanks to the built-in pull-down resistor, to high, when the button connected to 3V3 is pushed. A trigger of **`IRQ_FALLING`** would do the opposite: trigger the interrupt when the pin goes from high to low. In the case of your circuit, **`IRQ_RISING`**

will trigger as soon as the button is pushed; **IRQ_FALLING** would trigger only when the button is released afterwards.

THE RISE AND FALL OF IRQS

If you need to write a program which triggers an interrupt whenever a pin changes, without caring whether it's rising or falling, you can combine the two triggers using a pipe or vertical bar symbol (|):

```
button.irq(trigger=machine.Pin.IRQ_RISING | machine.Pin.IRQ_FALLING, handler=button_handler)
```

Your program should look like this:

```
import machine
import utime
import urandom

led = machine.Pin(15, machine.Pin.OUT)
button = machine.Pin(14, machine.Pin.IN, machine.Pin.PULL_DOWN)

def button_handler(pin):
    button.irq(handler=None)
    print(pin)

led.value(1)
utime.sleep(urandom.uniform(5, 10))
led.value(0)
button.irq(trigger=machine.Pin.IRQ_RISING, handler=button_handler)
```

Click the Run button and save the program to your Pico as **Reaction_Game.py**. You'll see the LED light up: that's your signal to get ready with your finger on the button. When the LED goes out, press the button as quickly as you can.

When you press the button, it triggers the handler code you wrote earlier. Look at the Shell area: you'll see your Pico has printed a message, confirming that the interrupt was triggered by pin GP14. You'll also see another detail: **mode=IN** tells you the pin was configured as an input. That message doesn't make for much of a game, though: for that, you need a way to time the player's reaction speed. Start by deleting the line **print(pin)** from your button handler – you don't need it any more.

Go to the bottom of your program and add a new line, just above where you set up the interrupt:

```
    timer_start = utime.ticks_ms()
```

This creates a new variable called **timer_start** and fills it with the output of the **utime.ticks_ms()** function, which counts the number of milliseconds that have elapsed since the utime library began counting. This provides a reference point: the time just after the LED went out and just before the interrupt trigger became ready to read the button press.
Next, go back to your button handler and add the following two lines, remembering that they'll need to be indented by four spaces so MicroPython knows they form part of the nested code:

```
        timer_reaction = utime.ticks_diff(utime.ticks_ms(), timer_start)
        print("Your reaction time was " + str(timer_reaction) +
" milliseconds!")
```

The first line creates another variable, this time for when the interrupt was actually triggered – in other words, when you pressed the button. Rather than simply taking a reading from **utime.ticks_ms()** as before, though, it uses **utime.ticks_diff()** – a function which provides the difference between when this line of code is triggered and the reference point held in the variable **timer_start**.

The second line prints the result, but uses *concatenation* to format it nicely. The first bit of text, or string, tells the user what the number that follows means; the **+** means that whatever comes next should be printed alongside that string. In this case, what comes next is the contents of the **timer_reaction** variable – the difference, in milliseconds, between when you took the reference point for the timer and when the button was pushed and the interrupt triggered.

Finally, the last line concatenates one more string so the user knows the number is measured in milliseconds, and not some other unit like seconds or microseconds. Pay attention to spacing: you'll see that there's a trailing space after 'was' and before the end quote of the first string, and a leading space after the open quote of the second string and before the word 'milliseconds'. Without those, the concatenated string will print something like 'Your reaction time was323milliseconds'.

Your program should now look like this:

```
import machine
import utime
import urandom

led = machine.Pin(15, machine.Pin.OUT)
button = machine.Pin(14, machine.Pin.IN, machine.Pin.PULL_DOWN)

def button_handler(pin):
    button.irq(handler=None)
    timer_reaction = utime.ticks_diff(utime.ticks_ms(), timer_start)
```

```
        print("Your reaction time was " + str(timer_reaction) +
" milliseconds!")

    led.value(1)
    utime.sleep(urandom.uniform(5, 10))
    led.value(0)
    timer_start = utime.ticks_ms()
    button.irq(trigger=machine.Pin.IRQ_RISING, handler=button_handler)
```

Click the Run button again, wait for the LED to go out, and push the button. This time, instead of a report on the pin which triggered the interrupt, you'll see a line telling you how quickly you pushed the button – a measurement of your reaction time. Click the Run button again and see if you can push the button more quickly this time – in this game, you're trying for as low a score as possible!

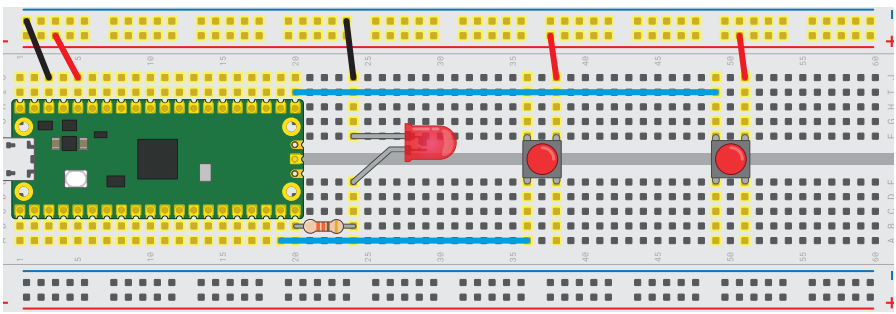
CHALLENGE: CUSTOMISATION

Can you tweak your game so that the LED stays lit for a longer time? What about staying lit for a shorter time? Can you personalise the message that prints to the Shell area, and add a second message congratulating the player?

A two-player game

Single-player games are fun, but getting your friends involved is even better. You can start by inviting them to play your game and comparing your high – or, rather, low – scores to see who has the quickest reaction time. Then, you can modify your game to let you go head-to-head!

Start by adding a second button to your circuit. Wire it up the same as the first button, with one leg going to the power rail of your breadboard but with the other going to pin GP16 – the pin across the board from GP14 where the LED is connected, at the opposite corner of your Pico.



▲ **Figure 6-2:** The circuit for a two-player reaction game

Make sure the two buttons are spaced far enough apart that each player has room to put their finger on their button. Your finished circuit should look like **Figure 6-2.**

Although your second button is now connected to your Pico, it doesn't know what to do with it yet. Go back to your program in Thonny and find where you set up the first button. Directly beneath this line, add:

```
right_button = machine.Pin(16, machine.Pin.IN, machine.Pin.PULL_DOWN)
```

You'll notice that the name now specifies which button you're working with: the right-hand button on the breadboard. To avoid confusion, edit the line above so that you make it clear what was the only button on the board is now the left-hand button:

```
left_button = machine.Pin(14, machine.Pin.IN, machine.Pin.PULL_DOWN)
```

You'll need to make the same change elsewhere in your program, too. Go to your button handler function and change the line:

```
button.irq(handler=None)
```

...So that it reads:

```
left_button.irq(handler=None)
```

Next, add a new line beneath it for the second button – remembering that, like all the code in your handler, it needs to be indented by four spaces so MicroPython knows it's part of the function:

```
right_button.irq(handler=None)
```

Scroll right down to the bottom of your program and change the line that sets up the interrupt trigger so it reads:

```
left_button.irq(trigger=machine.Pin.IRQ_RISING, handler=button_handler)
```

And, again, add another line beneath it to set up an interrupt trigger on your new button as well:

```
right_button.irq(trigger=machine.Pin.IRQ_RISING, handler=button_handler)
```

Your program should now look like this:

```
import machine
import utime
```

```
import urandom

led = machine.Pin(15, machine.Pin.OUT)
left_button = machine.Pin(14, machine.Pin.IN, machine.Pin.PULL_DOWN)
right_button = machine.Pin(16, machine.Pin.IN, machine.Pin.PULL_DOWN)

def button_handler(pin):
    left_button.irq(handler=None)
    right_button.irq(handler=None)
    timer_reaction = utime.ticks_diff(utime.ticks_ms(), timer_start)
    print("Your reaction time was " + str(timer_reaction) +
" milliseconds!")

led.value(1)
utime.sleep(urandom.uniform(5, 10))
led.value(0)
right_button.irq(trigger=machine.Pin.IRQ_RISING, handler=button_handler)
left_button.irq(trigger=machine.Pin.IRQ_RISING, handler=button_handler)
```

Click the Run icon, wait for the LED to go out, then press the left-hand push-button switch: you'll see that the game works the same as before, printing your reaction time to the Shell area. Click the Run icon again, but this time when the LED goes out, press the right-hand button: the game will work just the same, printing your reaction time as normal.

INTERRUPTS AND HANDLERS

Each interrupt you create needs a handler, but a single handler can deal with as many interrupts as you like. In the case of this program, you have two interrupts both going to the same handler – meaning that whichever interrupt triggers, they'll run the same code. A different program might have two handlers, letting each interrupt run different code – it all depends on what you need your program to do.

To make the game a little more exciting, you can have it report on which of the two players was the first to press the button. Go back to the top of your program, just below where you set up the LED and the two buttons, and add the following:

```
fastest_button = None
```

This sets up a new variable, **fastest_button**, and sets its initial value to **None** – because no button has yet been pressed. Next, go to the bottom of your button handler and delete the two lines which handle the timer and printing – then replace them with:

```
    global fastest_button
    fastest_button = pin
```

Remember that these lines will need to be indented by four spaces so that MicroPython knows they're part of the function. These two lines allow your function to change, rather than just read, the **fastest_button** variable, and set it to contain the details of the pin which triggered the interrupt – the same details your game printed to the Shell area earlier in the chapter, including the number of the triggering pin.

Now go right to the bottom of your program, and add these two new lines:

```
while fastest_button is None:
    utime.sleep(1)
```

This creates a loop, but it's not an infinite loop: here, you've told MicroPython to run the code in the loop only when the **fastest_button** variable is still zero – the value it was initialised with at the start of the program. In effect, this pauses your program's main thread until the interrupt handler changes the value of the variable. If neither player presses a button, the program will simply pause.

Finally, you need a way to determine which player won – and to congratulate them. Type the following at the bottom of the program, making sure to delete the four-space indent Thonny will have created for you on the first line – these lines do not form part of the loop:

```
if fastest_button is left_button:
    print("Left Player wins!")
elif fastest_button is right_button:
    print("Right Player wins!")
```

The first line sets up an 'if' conditional which looks to see if the **fastest_button** variable is **left_button** – meaning the IRQ was triggered by the left-hand button. If so, it will print a message – with the line below indented by four spaces so that MicroPython knows it should run it only if the conditional is true – congratulating the left-hand player, whose button is connected to GP14.

The next line, which should not be indented, extends the conditional as an 'elif' – short for 'else if', a way of saying 'if the first conditional wasn't true, check this conditional next'. This time it looks to see if the **fastest_button** variable is **right_button** – and, if so, prints a message congratulating the right-hand player, whose button is connected to GP16.

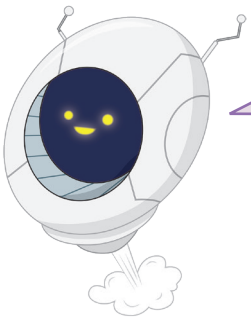
Your finished program should look like this:

```
import machine
import utime
import urandom

led = machine.Pin(15, machine.Pin.OUT)
left_button = machine.Pin(14, machine.Pin.IN, machine.Pin.PULL_DOWN)
right_button = machine.Pin(16, machine.Pin.IN, machine.Pin.PULL_DOWN)

fastest_button = None

def button_handler(pin):
    left_button.irq(handler=None)
    right_button.irq(handler=None)
    global fastest_button
    fastest_button = pin

led.value(1)
utime.sleep(urandom.uniform(5, 10))
led.value(0)
left_button.irq(trigger=machine.Pin.IRQ_RISING, handler=button_handler)
right_button.irq(trigger=machine.Pin.IRQ_RISING, handler=button_handler)
while fastest_button is None:
    utime.sleep(1)
if fastest_button is left_button:
    print("Left Player wins!")
elif fastest_button is right_button:
    print("Right Player wins!")
```

Press the Run button and wait for the LED to go out – but don't press either of the push-button switches just yet. You'll see that the Shell area remains blank, and doesn't bring back the >>> prompt; that's because the main thread is still running, sitting in the loop you created.

Now push the left-hand button, connected to pin GP14. You'll see a message congratulating you printed to the Shell – your left hand was the winner! Click Run again and try pushing the right-hand button after the LED goes out: you'll see another message printed, this time congratulating your right hand. Click Run again, this time with one finger on each button: push them both at the same time and see whether your right hand or left hand is faster!

Now that you've created a two-player game, you can invite your friends to play along and see which of you has the fastest reaction times!



CHALLENGE: TIMINGS

Can you modify the messages that print? Can you add a third button, so that three people can play at once? Is there an upper limit to how many buttons you could add? Can you add the timer back into your program, so it tells the winning player how quick their reaction time was?

Chapter 7
Burglar alarm

Use a motion sensor to detect intruders and sound the alarm with a flashing light and siren

Another real-world use of microcontrollers is in alarm systems. From the alarm clock that gets you up in the morning to fire alarms, burglar alarms, and even the alarms that sound when there's a problem at a nuclear power station, microcontrollers help keep us all safe.

In this chapter you're going to build your own burglar alarm, which works in exactly the same way as a commercial version: a special motion sensor keeps watch for anyone entering the room when they shouldn't be there, and flashes a light while sounding a siren to alert people to the intrusion. Whether you're protecting a bank vault or just trying to keep spying siblings out of your room, a burglar alarm is sure to come in handy.

For this project you'll need your Pico; a breadboard; an LED of any colour; a 330 Ω resistor; an active piezoelectric buzzer; one or more HC-SR501 passive infrared (PIR) sensors; and a selection of male-to-male (M2M) and male-to-female (M2F) jumper wires. You'll also need a micro USB cable, and to connect your Pico to your Raspberry Pi or other computer running the Thonny MicroPython IDE.

The HC-SR501 PIR sensor

In previous chapters, you've been working with simple input components in the form of push-button switches. This time, you're going to be using a specialised input known as a *passive infrared sensor* or *PIR*. There are hundreds of different PIR sensors available; the HC-SR501 is low-cost, high-performance, and works perfectly with your Pico.

If you've picked up a different model of sensor, look at its documentation to double-check which pins are which; also make sure that it operates at a 3V3 logic level, just like your Pico – if you connect a sensor which uses a higher voltage, such as a 12 V sensor, it will damage your Pico beyond repair. Some sensors may need a small switch or a jumper changing to move between logic voltage levels; this will be noted in the documentation.

A passive infrared sensor is designed to detect movement, in particular movement from people and other living things. It works a little like a camera, but instead of capturing visible light it looks for the heat emitted from a living body as infrared radiation. It's known as a passive infrared sensor, rather than active, because just like a camera sensor it doesn't send out any infrared signals of its own.

The actual sensor is buried underneath a plastic lens, typically shaped like a half-ball. The lens isn't technically necessary for the sensor to work, but serves to provide a wider *field of vision (FOV)*; without the lens, the PIR sensor would only be able to see movement in a very narrow angle directly in front of the sensor. The lens serves to pull in infrared from a much wider angle, meaning a single PIR sensor is able to watch for movement over the majority of a room.

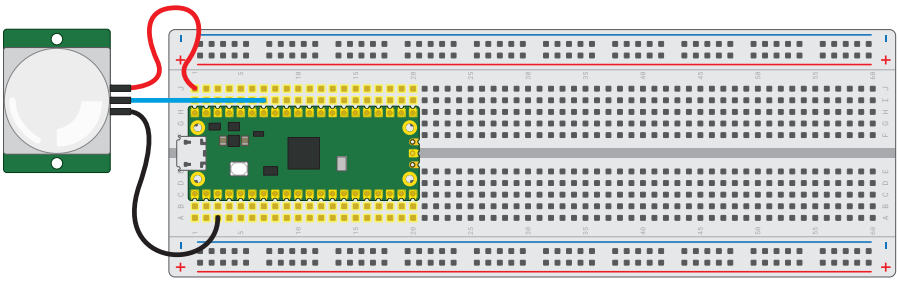
In commercial burglar alarm systems, a PIR sensor is only one of the sensors used; others include break-glass sensors which can tell when a window has been smashed, magnetic sensors which monitor whether a door is open or closed, acoustic sensors which can pick up a burglar's footsteps, and vibration sensors for telling if a lock is being forced open. A simple PIR sensor, though, is often enough for a low-security area – think a reception room, rather than a bank vault.

Pick up your HC-SR501 sensor now and take a look at it. The first thing to notice is that it has a circuit board of its own, a lot like your Pico – only smaller. As well as the sensor and lens, there are several other components: a small black *integrated circuit (IC)* which drives the sensor, some *capacitors*, and tiny surface-mount resistors. You may also see some small *potentiometers* which you can twist with a screwdriver to adjust the sensitivity of the sensor and how long it stays active when triggered; leave these as they are for now.

You'll also see three male pins, exactly like the pins on the bottom of your Pico. You can't push these directly into your breadboard, though, as the components on the board will get in the way. Instead, take three male-to-female (M2F) jumper wires and insert the female ends onto the pins on your HC-SR501.

Next, take the male ends and wire them to the breadboard and your Pico. You'll need to check the documentation for your sensor when wiring it to your Pico: a lot of different companies make HC-SR501 sensors, and they don't always use the same pins for the same purposes. For the sensor illustrated in **Figure 7-1** (overleaf), the pins are set up so that the

ground (GND) pin is on the bottom, the signal or trigger pin is in the middle, and the power pin is on the right; your sensor may need the wires placing in a different order!



▲ **Figure 7-1:** Wiring an HC-SR501 PIR sensor to your Pico

Start with the ground wire: this needs to be connected to any of your Pico's ground pins. In **Figure 7-1**, it's connected to the first ground pin in breadboard row 3. Next, connect the signal wire: this should be connected to your Pico's GPIO pin GP28.

Finally, you need to connect the power wire. Don't connect this to your Pico's 3V3 pin, though: the HC-SR501 is a 5 V device, meaning that it needs five volts of electricity in order to work. If you wire the sensor to your Pico's 3V3 pin, it won't work – the pin simply doesn't provide enough power.

To give your sensor the 5 V power it needs, wire it to the very top-right pin of your Pico – VBUS. This pin is connected to the micro USB port on your Pico, and taps into the USB 5 V power line before it's converted to 3.3 V to run your Pico's microprocessor. All three HC-SR501 pins should now be wired to your Pico: ground, signal, and power.

Programming your alarm

You'll need to program your Pico to recognise the sensor. Handily, this is no more difficult than reading a button – in fact, you can use the very same code. Start by creating a new program and importing the machine library so you can configure your Pico's GPIO pin:

```
import machine
```

Then set up the pin you wired your HC-SR501 sensor to, GP28:

```
sensor_pir = machine.Pin(28, machine.Pin.IN, machine.Pin.PULL_DOWN)
```

Like the reaction game you made, a burglar alarm's inputs should act as an interrupt – stop the program doing whatever it was doing and react whenever the sensor is triggered. As before, start by defining a callback function to handle the interrupt:

```
def pir_handler(pin):
```

```
    print("ALARM! Motion detected!")
```

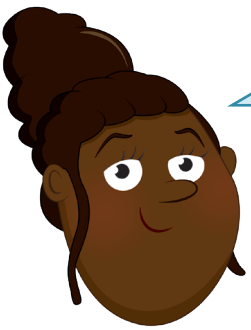
Remember that the second line needs to be indented by four spaces so MicroPython knows it's part of the function; Thonny will put these spaces in automatically when you press **ENTER** after the first line.

Finally, set up the interrupt itself. Remember that this isn't part of the handler function, so you'll need to delete the four spaces Thonny automatically inserts:

```
sensor_pir.irq(trigger=machine.Pin.IRQ_RISING, handler=pir_handler)
```

That's enough for the moment: remember that interrupts stay active regardless of what the rest of the program is doing, so there's no need to add an infinite loop to keep your program running. Click the Run icon and save the program to your Pico as **Burglar_Alarm.py**.

Wave your hand over the PIR sensor: a message will print to the Shell area confirming that the sensor saw you. If you keep waving your hand, the message will keep printing – but with a delay between each time it's printed.



NARROWING THE FOV

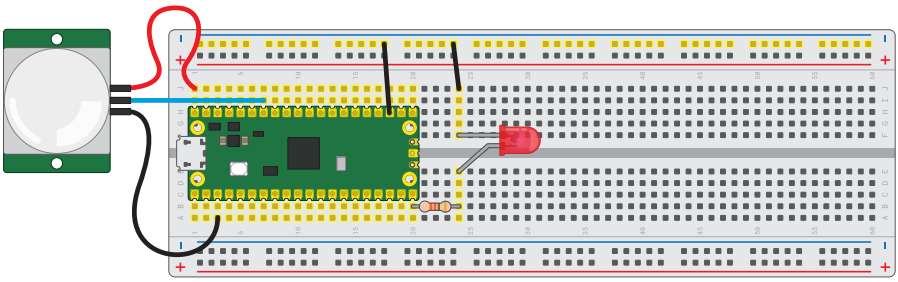
PIR sensors are designed to cover as wide a field of vision (FOV) as possible so that burglars can't simply scoot around the edges of a room. If you find your sensor is triggering when you don't want it to, there's a simple way to fix it: get the cardboard inner tube from a toilet roll and place the sensor at the bottom. The tube will act like horse blinkers, stopping the sensor from seeing things to the side so it can concentrate on things further ahead.

That delay isn't part of your program, but built into the HC-SR501 hardware itself: the sensor sends the trigger signal to your Pico's GPIO pin when motion is detected, and keeps that signal switched on for several seconds before dropping it. On most HC-SR501 sensors, that delay is adjusted using one of the small potentiometers: you can insert a screwdriver and turn it one way to decrease the delay and the other way to increase the delay. Check your sensor's documentation for which potentiometer controls the delay.

Because your interrupt trigger is set to fire on the rising edge of the signal, the message is printed as soon as the PIR sensor sends its signal of 1 or 'high'. Even if more motion is detected, the interrupt won't fire again until the built-in delay has passed and the signal has returned to 0, or 'low'.

Printing a message to the Shell is enough to prove your sensor is working, but it doesn't make for much of an alarm. Real burglar alarms have lights and sirens that alert everyone around that something's wrong – and you can add the same to your own alarm.

Start by wiring an LED, of any colour, to your Pico as shown in **Figure 7-2**. The longer leg, the anode, needs to connect to pin GP15 via a 330 Ω resistor – remember that without this resistor in place to limit the amount of current passing through the LED, you can damage both the LED and your Pico. The shorter leg, the cathode, needs to be wired to one of your Pico's ground pins – use your breadboard's ground rail and two male-to-male (M2M) jumper wires for this.



▲ **Figure 7-2:** Adding an LED to the burglar alarm

This time, you're going to handle delays in your program rather than relying on the delay built into the PIR sensor. Go to the top of your program and, just below the line **`import machine`**, add the following:

```
import utime
```

Next, add a new line just below where you set up the PIR sensor's pin:

```
led = machine.Pin(15, machine.Pin.OUT)
```

That's enough to configure the LED, but you'll need to make it light up. Add the following new line to your interrupt handler function – remembering that, like all the lines in the function, it will need to be indented by four spaces so MicroPython knows it's part of the nested code:

```
for i in range(50):
```

Press **ENTER** at the end of this line and you'll notice Thonny has automatically added another four spaces to make an eight-space indentation. That's because you've just created a finite loop, one which will run 50 times. The letter **`i`** represents an *increment*, a value which goes up each time the loop runs, and which is populated by the instruction **`range(50)`**.

Give your new loop something to do, remembering that these lines will need to be indented by eight spaces – which Thonny will have done automatically – as they form both part of the loop you just opened and the interrupt handler function:

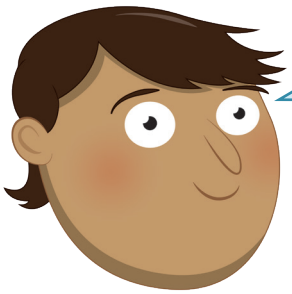
```
led.toggle()
```

```
utime.sleep_ms(100)
```

These two instructions are a little different than you've been using in earlier chapters. The first is a feature of the machine library which lets you simply change the value of an output pin, rather than set a value – so if the pin is currently 1, or high, toggling it will set it to 0, or low; if the pin is already 0, or low, toggling it will set it to high.

The second line uses the utime library to insert a pause in the program, but instead of being measured in seconds it's measured in milliseconds. The **`utime.sleep_ms()`** function is perfect for short delays of less than a second; if you're pausing for a second or more, stick with **`utime.sleep()`**.

Taken together, these two lines of code flash the LED on and off with a 100-millisecond – a tenth of a second – delay. The result is exactly the same as the LED blinking program you wrote back in **Chapter 4**, but whereas your previous program needed four lines of code – one to turn the LED one, another to pause, one to turn the LED off again, and a final pause before beginning the loop again – this only needs two.



VALUE VS TOGGLE

There are times when using the toggle function over setting a value makes sense, like when blinking an LED – but make sure you've thought through what you're trying to achieve first. If your project hinges on an output definitely being on or off at a given time – such as a warning light, or a pump which drains a water tank – always explicitly set the value rather than relying on a toggle.

Your program will now look like this:

```
import machine
import utime

sensor_pir = machine.Pin(28, machine.Pin.IN, machine.Pin.PULL_DOWN)
led = machine.Pin(15, machine.Pin.OUT)

def pir_handler(pin):
    print("ALARM! Motion detected!")
    for i in range(50):
        led.toggle()
        utime.sleep_ms(100)

sensor_pir.irq(trigger=machine.Pin.IRQ_RISING, handler=pir_handler)
```

Click the Run button, then wave your hand over the PIR sensor again: you'll see the usual alarm message print to the Shell area, then the LED will begin rapidly flashing as a visual alert. Wait for the LED to stop flashing, then wave your hand over the PIR sensor again: the message will print again, and the LED will repeat its flashing pattern.

To make your burglar alarm even more of a deterrent, you can make it flash slowly even when there's no motion being detected – warning would-be intruders that your room is under observation. Go to the very bottom of your program and add in the following lines:

```
while True:
    led.toggle()
    utime.sleep(5)
```

Click Run again, but leave the PIR sensor alone: you'll see the LED is now turning on for five seconds, then turning off for five seconds. This pattern will continue as long as the sensor isn't triggered; wave your hand over the PIR sensor and you'll see the LED rapidly flashing again, before going back to its slow-flash pattern.

This highlights a key difference between threads and interrupts: if you'd written this program using threads, your program would still be trying to toggle the LED on and off on a five-second interval even while your PIR handler is flashing it on and off on a 100-millisecond interval. That's because threads run concurrently, side by side.

With interrupts, the main program is paused while the interrupt handler runs – so the five-second toggle code you've written stops until the handler has finished flashing the LED, then picks up from where it left off. Whether you need your code to pause or to carry on running is the key to whether you need to use threads or interrupts, and will depend on exactly what your project is trying to do.

Inputs and outputs: putting it all together

Your burglar alarm now has a flashing LED to warn intruders away, and a way to see when it's been triggered without having to watch the Shell area for a message. Now all it needs is a siren – or, at least, a piezoelectric buzzer, which makes sound without deafening your neighbours.

Depending on which model you purchased, your piezoelectric buzzer will have either pins sticking out of the bottom or short wires attached to its sides. If the buzzer has pins, insert these into your breadboard so the buzzer is straddling the centre divide; if it has wires, place these in the breadboard and simply rest the buzzer on the breadboard.

If your buzzer has long enough wires, you might be able to connect them to the breadboard right next to your Pico's GPIO pins; if not, use male-to-male (M2M) jumper wires to wire the buzzer as shown in **Figure 7-3**. The red wire, or the positive pin marked with a + symbol, should be connected to pin GP14 at the bottom-left of your Pico, just above the pin you're using for the LED. The black wire, or the negative pin marked with a minus (-) symbol or the letters GND, needs to be connected to the ground rail of your breadboard.

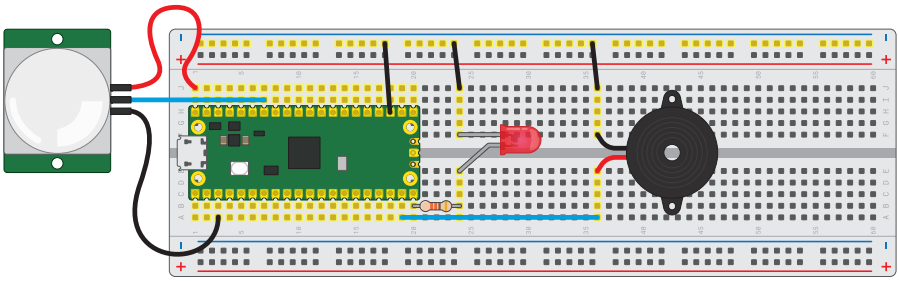


Figure 7-3: Wiring a two-wire piezoelectric buzzer

If your buzzer has three pins, connect the leg marked with a minus symbol (-) or the letters GND to the ground rail of your breadboard, the pin marked with S or SIGNAL to pin GP14 on your Pico, and the remaining leg – which is usually the middle leg – to the 3V3 pin on your Pico.

If you run your program now, nothing will change: the buzzer will only make a sound when it receives power from your Pico's GPIO pins. Go back to the top of your program and set the buzzer up just below where you set the LED up:

```
buzzer = machine.Pin(14, machine.Pin.OUT)
```

Next, change your interrupt handler to add a new line below `led.toggle()` – remembering that, as it's part of both the loop and the handler function, it will need to be indented by eight spaces:

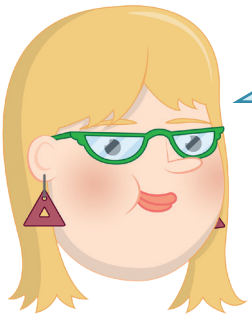
```
buzzer.toggle()
```

Your program will now look like this:

```
import machine
import utime

sensor_pir = machine.Pin(28, machine.Pin.IN, machine.Pin.PULL_DOWN)
led = machine.Pin(15, machine.Pin.OUT)
buzzer = machine.Pin(14, machine.Pin.OUT)

def pir_handler(pin):
    print("ALARM! Motion detected!")
    for i in range(50):
        led.toggle()
        buzzer.toggle()
        utime.sleep_ms(100)

sensor_pir.irq(trigger=machine.Pin.IRQ_RISING, handler=pir_handler)
```

```
while True:
    led.toggle()
    utime.sleep(5)
```

Click Run and wave your hand over the PIR sensor: the LED will flash rapidly, as before, but this time it'll be accompanied by a beeping sound from the buzzer. Congratulations: that should be more than enough to scare an intruder away from ransacking your secret stash of sweets!



WARNING

When using an active buzzer, it will continue to sound for as long as the pin it's connected to is high – in other words, has a value of 1. Because your loop runs an even number of times, it finishes with the buzzer switched off; change the loop to run an odd number of times, though, and it will finish with the buzzer still sounding – and pressing the Stop button won't turn it off. If this happens, simply unplug your Pico's micro USB cable and plug it back in again – then change your program so it doesn't happen again!

If you find your buzzer is clicking, rather than beeping, then you're using a *passive buzzer* rather than an *active buzzer*. An active buzzer has a component inside known as an *oscillator,* which rapidly moves the metal plate to make the buzzing sound; a passive buzzer lacks this component, meaning you need to replace it with some code of your own.

If you're using a passive buzzer, try this version of the program instead – it toggles the pin connected to the buzzer on and off very quickly, mimicking the effect of the oscillator in an active buzzer:

```
import machine
import utime

sensor_pir = machine.Pin(28, machine.Pin.IN, machine.Pin.PULL_DOWN)
led = machine.Pin(15, machine.Pin.OUT)
buzzer = machine.Pin(14, machine.Pin.OUT)

def pir_handler(pin):
    print("ALARM! Motion detected!")
    for i in range(50):
        led.toggle()
        for j in range(25):
            buzzer.toggle()
```

```
            utime.sleep_ms(3)

sensor_pir.irq(trigger=machine.Pin.IRQ_RISING, handler=pir_handler)

while True:
    led.toggle()
    utime.sleep(5)
```

Note that the new loop, which controls the buzzer, doesn't use the letter `i` to track the increment; that's because you're already using that letter for the outer loop – so it uses the letter `j`. The short delay of just three milliseconds, meanwhile, means the pin connected to the buzzer turns on and off rapidly enough for it to make a buzzing noise.

Try changing the delay to four milliseconds instead of three and you'll find the buzzer sounds at a lower pitch. Changing the delay changes the frequency of the buzzer's oscillation: a longer delay means it oscillates at a lower frequency, making it a lower-pitched sound; a shorter delay makes it oscillate at a higher frequency, making it a higher-pitched sound.

Extending your alarm

Burglar alarms rarely cover a single room: instead, they use a network of multiple sensors to monitor multiple rooms from a single alarm system. Your Pico-based burglar alarm can work in exactly the same way, adding in multiple sensors to cover multiple areas at once.

You'll need an HC-SR501 sensor for each area you want to cover; in this example you'll add one more sensor for a total of two, but you can keep going and add as many sensors as you need.

Both of your sensors need 5 V power to work, but you've already used the VUSB pin on your Pico for the first sensor. If your breadboard has enough room, you could put a male-to-female (M2F) jumper wire next to the one connected to your first sensor and use it for the second; a neater approach, though, is to use the power rail on your breadboard.

Disconnect the first sensor's power wire from the breadboard end, and insert it into the power rail coloured red or marked with a plus (+) symbol. Take a male-to-male (M2M) jumper wire and connect the same power rail to your Pico's VUSB pin. Next, take a male-to-female (M2F) jumper wire and connect the power rail to your second PIR sensor's power input pin.

Finally, wire up the ground pin and signal pin of your second PIR sensor as before – but this time connect the signal pin to pin GP22 on your Pico, as shown in **Figure 7-4** (overleaf). Your circuit now has two sensors, each connected to a separate pin.

Setting your program up to read the second sensor as well as the first is as simple as adding two new lines. Start by setting the second sensor up, adding a new line below where you set the first sensor up:

```
sensor_pir2 = machine.Pin(22, machine.Pin.IN, machine.Pin.PULL_DOWN)
```

Then create a new interrupt, again directly beneath your first interrupt:

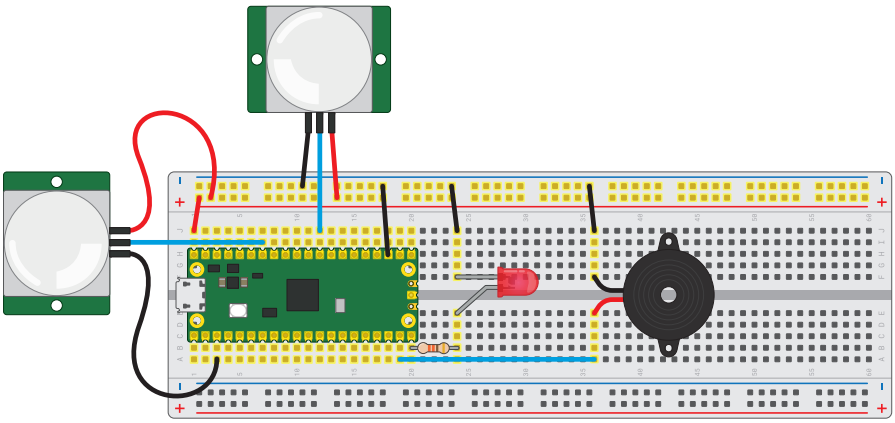


Figure 7-4: Adding a second PIR sensor to cover another room

```
sensor_pir2.irq(trigger=machine.Pin.IRQ_RISING, handler=pir_handler)
```

Remember that you can have multiple interrupts with a single handler, so there's no need to change that part of your program.

Click Run, and wave your hand over the first PIR sensor: you'll see the alert message, the LED flash, and the buzzer sound as normal. Wait for them to finish, then wave your hand over the second PIR sensor: you'll see your burglar alarm respond in exactly the same way.

To make your alarm really smart, you can customise the message depending on which pin was responsible for the interrupt – and it works exactly the same way as in the two-player reaction game you wrote earlier.

Go back to your interrupt handler and modify it so it looks like:

```
def pir_handler(pin):
    if pin is sensor_pir:
        print("ALARM! Motion detected in bedroom!")
    elif pin is sensor_pir2:
        print("ALARM! Motion detected in living room!")
    for i in range(50):
        led.toggle()
        buzzer.toggle()
        utime.sleep_ms(100)
```

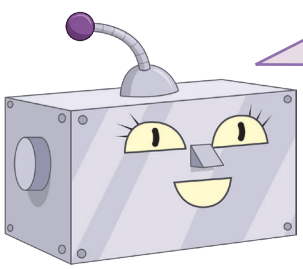
Just as in the reaction game project in **Chapter 6,** this code uses the fact that an interrupt reports which pin it was triggered by: if the PIR sensor attached to pin GP28 is responsible, it will print one message; if it was the PIR sensor attached to pin GP22, it will print another.

Your finished program will look like this:

```
import machine
import utime

sensor_pir = machine.Pin(28, machine.Pin.IN, machine.Pin.PULL_DOWN)
sensor_pir2 = machine.Pin(22, machine.Pin.IN, machine.Pin.PULL_DOWN)
led = machine.Pin(15, machine.Pin.OUT)
buzzer = machine.Pin(14, machine.Pin.OUT)

def pir_handler(pin):
    if pin is sensor_pir:
        print("ALARM! Motion detected in bedroom!")
    elif pin is sensor_pir2:
        print("ALARM! Motion detected in living room!")
    for i in range(50):
        led.toggle()
        buzzer.toggle()
        utime.sleep_ms(100)

sensor_pir.irq(trigger=machine.Pin.IRQ_RISING, handler=pir_handler)
sensor_pir2.irq(trigger=machine.Pin.IRQ_RISING, handler=pir_handler)

while True:
    led.toggle()
    utime.sleep(5)
```

If you're using a passive, rather than active, buzzer, remember you'll need to change the buzzer toggle to a loop in order to have it beep. Click Run and wave your hand over one sensor, then the other, to see both messages print to the Shell area.

Congratulations: you now know how to build a modular burglar alarm capable of covering as many areas as you need!

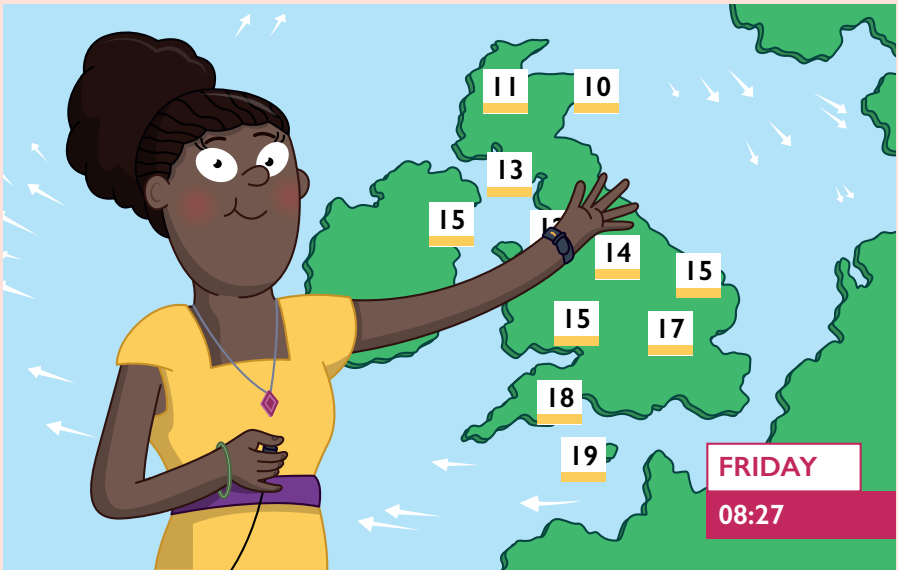


CHALLENGE: CUSTOMISATION

Can you extend the burglar alarm with another PIR sensor? What about adding another LED, or another buzzer? Can you change the messages that print to match the areas you're covering with each sensor? Can you make the buzzer sound for longer, or for less time? Can you think of any other sensors, apart from a PIR sensor, that might work well in a burglar alarm?

Chapter 8
Temperature gauge

Use your Raspberry Pi Pico's built-in ADC to convert analogue inputs, and read its internal temperature sensor



In previous chapters you've been using the digital inputs on your Raspberry Pi Pico. A digital input is either on or off, a *binary* state. When a push-button switch is pressed, it changes a pin from low, off, to high, on; when a passive infrared sensor detects motion, it does the same.

Your Pico can accept another type of input signal, though: *analogue input*. Whereas digital is only ever either on or off, an analogue signal can be anything from completely off to completely on – a range of possible values. Analogue inputs are used for everything from volume controls to gas, humidity, and temperature sensors – and they work through a piece of hardware known as an *analogue-to-digital converter (ADC)*.

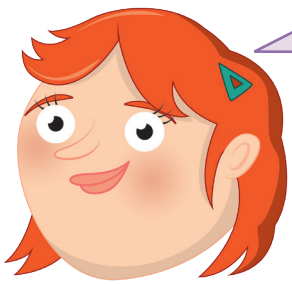
In this chapter you'll learn how to use the ADC on your Pico – and how to tap in to its internal temperature sensor to build a data-logging heat measurement gadget. You'll also learn a technique for creating an analogue-like output. For this you'll need your Pico; an LED of any colour and 330 Ω resistor; a 10 kΩ potentiometer; and a selection of male-to-male (M2M) jumper wires. You'll also need a micro USB cable, and to connect your Pico to your Raspberry Pi or other computer running the Thonny MicroPython IDE.

The analogue-to-digital converter

Raspberry Pi Pico's RP2040 microcontroller is a digital device, like all mainstream microcontrollers: it is built up of thousands of *transistors*, tiny switch-like devices which are either on or off. As a result, there's no way for your Pico to truly understand an analogue signal – one which can be anything on a spectrum between fully off and fully on – without relying on an additional piece of hardware: the analogue-to-digital converter (ADC).

As the name suggests, an analogue-to-digital converter takes an analogue signal and changes it to a digital one. You won't see the ADC on your Pico, no matter how closely you look: it's built into RP2040 itself. Many microcontrollers have their own ADCs, just like RP2040, and the ones that don't can use an external ADC connected to one or more of their digital inputs.

An ADC has two key features: its *resolution*, measured in digital bits, and its *channels*, or how many analogue signals it can accept and convert at once. The ADC in your Pico has a resolution of 12 bits, meaning that it can transform an analogue signal into a digital signal as a number ranging from 0 to 4095 – though this is handled in MicroPython transformed to a 16-bit number ranging from 0 to 65,535, so that it behaves the same as the ADC on other MicroPython microcontrollers. It has three channels brought out to the GPIO pins: GP26, GP27, and GP28, which are also known as GP26_ADC0, GP27_ADC1, and GP28_ADC2 for analogue channels 0, 1, and 2. There's also a fourth ADC channel, which is connected to a temperature sensor built into RP2040; you'll find out more about that later in the chapter.



WHY 65,535?

The number '65,535' looks strange at first glance – why that, and why not simply 0–100? The answer ties in to the fact your Pico works on a binary number system, where the only possible values for a digit are 0 or 1. A 16-bit binary number is made up of 16 digits, and the maximum possible value is 16 ones: 1111111111111111. If you convert that back into decimal numbers, the 0–9 counting system humans use, you get 65,535.

Reading a potentiometer

Every pin connected to your Pico's analogue-to-digital converter can also be used as a simple digital input or output; to use it as an analogue input, you'll need an analogue signal – and you can easily make one with a potentiometer.

There are various types of potentiometer available: some, like the ones on the HC-SR501 passive infrared sensor you used in **Chapter 7**, are designed to be adjusted with a screwdriver; others, often used for volume controls and other inputs, have knobs or sliders. The most common type has a small, usually plastic, knob coming out of the top or front: this is known as a *rotary potentiometer.*

Pick up your potentiometer and turn it over: you'll see it has three pins which fit in the breadboard. Depending on how you wire these pins up, the potentiometer works in two different ways. Start by inserting the potentiometer into your breadboard, being careful not to bend the pins. Wire the middle pin to pin GP26_ADC0 on your Pico using a male-to-male (M2M) jumper wire, as shown in **Figure 8-1** – if your Pico is inserted into the breadboard at the very top, it'll be on row 10. Finally, take two more jumper wires and wire one of the potentiometer's outer pins – it doesn't matter which – to your breadboard's power rail and the power rail to your Pico's 3V3 pin.

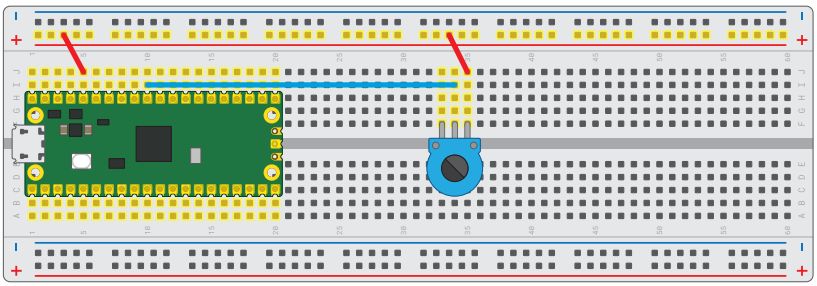


Figure 8-1: A potentiometer wired with two pins connected

Open Thonny and begin a new program:

```
import machine
import utime
```

Like the digital general-purpose input/output (GPIO) pins, the analogue input pins are handled by the machine library – and just like the digital pins, they need to be set up before you can use them. Continue your program:

```
potentiometer = machine.ADC(26)
```

This configures pin GP26_ADC0 as the first channel, ADC0, on the analogue-to-digital converter. To read from the pin, set up a loop:

```
while True:
    print(potentiometer.read_u16())
    utime.sleep(2)
```

In this loop, reading the value of the pin and printing it take place on a single line: this is a more compact alternative to reading the value into a variable and then printing the variable, but only works if you don't want to do anything with the reading other than print it – which is exactly what this program needs at the moment.

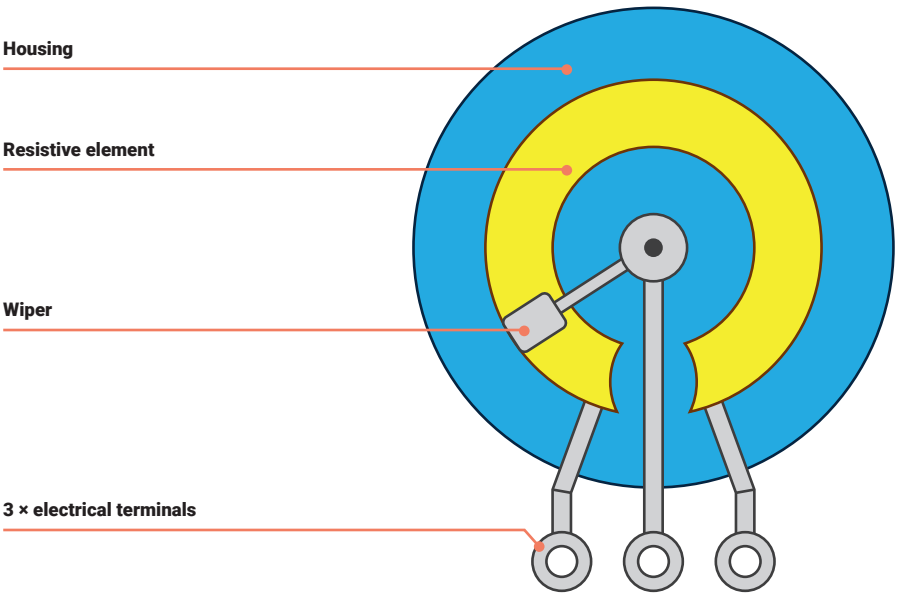
Reading an analogue input is almost identical to reading a digital input, except for one thing: when reading a digital input you use `read()`, but this analogue input is read with `read_u16()`. That last part, `u16`, simply warns you that rather than receiving a binary 0 or 1 result, you'll receive an *unsigned 16-bit integer* – a whole number between 0 and 65,535.

Click the Run icon and save your program as **Potentiometer.py.** Watch the Shell: you'll see your program print out a large number, likely over 60,000. Try turning the potentiometer all the way in one direction: depending on the direction you turned the knob and the outer leg you used in your circuit, the number will go up or down. Turn it the other way: the value will change in the opposite direction.

No matter which way you turn it, though, it will never get anywhere near 0. That's because with only two legs connected, the potentiometer is acting as a component known as a *variable resistor* or *varistor*. A varistor is a resistor with a value you can change – in the case of a 10 kΩ potentiometer, between 0 Ω and 10,000 Ω. The higher the resistance, the less voltage from the 3V3 pin reaches your analogue input – so the number goes down. The lower the resistance, the more voltage reaches your analogue input – so the number goes up.

A potentiometer works by having a conductive strip inside, connected to the two outer pins, and a *wiper* or *brush* connected to the inner pin (**Figure 8-2**). As you turn the knob, the wiper moves closer to one end of the strip and further away from the other. The further the wiper gets from the end of the strip you wired to your Pico's 3V3 pin, the higher the resistance; the closer it gets, the lower the resistance.

Varistors are extremely useful components, but there's a drawback: you'll notice no matter how far you turn the knob in either direction, you can never get a value of 0 – or anywhere close

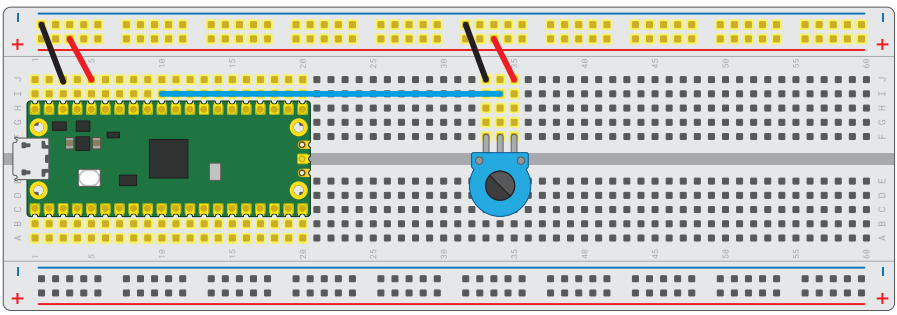


▲ **Figure 8-2:** How a potentiometer works

to it. That's because a 10 kΩ resistor isn't strong enough to drop the 3V3 pin's output to 0 V. You could look for a bigger potentiometer with a higher maximum resistance, or you could simply wire your existing potentiometer up as a *voltage divider*.

A potentiometer as a voltage divider

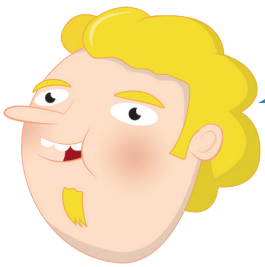
The unused pin on your potentiometer isn't there for show: adding a connection to that pin to your circuit completely changes how the potentiometer works. Click the Stop icon to stop your program, and grab two male-to-male (M2M) jumper wires. Use one to connect the unused pin of your potentiometer to your breadboard's ground rail as shown in **Figure 8-3**. Take the other and connect the ground rail to a GND pin on your Pico.



▲ **Figure 8-3:** Wiring the potentiometer as a voltage divider

Click the Run icon to restart your program. Turn the potentiometer knob again, all the way one direction then all the way the other. Watch the values that are printed to the Shell area: unlike before, they're now going from near-zero to nearly a full 65,535 – but why?

Adding the ground connection to the other end of the potentiometer's conductive strip has created a voltage divider: whereas before the potentiometer was simply acting as a resistor between the 3V3 pin and the analogue input pin, it's now dividing the voltage between the 3.3 V output by the 3V3 pin and the 0 V of the GND pin. Turn the knob fully one direction, you'll get 100 percent of the 3.3V; turn it fully the other way, 0 percent.



ZERO'S THE HARDEST NUMBER

If you can't get your Pico's analogue input to read exactly zero or exactly 65,535, don't worry – you haven't done anything wrong! All electronic components are built with a *tolerance*, which means any claimed value isn't going to be precise. In the case of the potentiometer, it will likely never reach exactly 0 or 100 percent of its input – but it will get you very close!

The number you see printed to the Shell is a decimal representation of the raw output of the analogue-to-digital converter – but it's not the friendliest way to see it, especially if you forget that 65,535 means 'full voltage'.

There's an easy way to fix that, though: a simple mathematical equation. Go back to your program, and add the following above your loop:

```
conversion_factor = 3.3 / (65535)
```

This sets up a mathematical way to convert the number that the analogue-to-digital converter gives you into a fair approximation of the actual voltage it represents. The first number is the maximum possible voltage that the pin can expect: 3.3 V, from your Pico's 3V3 pin; the second number is the maximum value the analogue input reading can be, 65,535.

Taken all together, the conversion factor is a number created by '3.3 divided by 65,535' – the maximum possible voltage divided by the range of values the analogue-to-digital converter reports, which is in turn a feature of its resolution in bits.

With your conversion factor set up, you simply need to use it in your program. Go back to your loop, and edit it to read:

```
while True:
    voltage = potentiometer.read_u16() * conversion_factor
    print(voltage)
    utime.sleep(2)
```

The first line inside the loop takes a reading from the potentiometer via the analogue input pin, and multiplies it – the * symbol – by the conversion factor you set up earlier in the program, storing the result as the variable voltage. That variable is then printed to the Shell, in place of the raw reading you used earlier.

Your finished program will look like this:

```
import machine
import utime

potentiometer = machine.ADC(28)

conversion_factor = 3.3 / (65535)

while True:
    voltage = potentiometer.read_u16() * conversion_factor
    print(voltage)
    utime.sleep(2)
```

LINEAR VS LOG

If you find turning your potentiometer slowly between one limit and the other makes the numbers change slowly at first then start changing more rapidly, or the other way around, you're almost certainly using a *logarithmic* or *log potentiometer*. Whereas a *linear potentiometer* changes smoothly across its entire range, a log potentiometer starts off making small changes, then rapidly ramps the speed of change up. Log potentiometers are commonly used for volume controls on amplifiers, whereas linear potentiometers are more common for microcontroller-based devices like your Pico.

Click the Run icon. Turn the potentiometer all the way in one direction, then the other. Watch the numbers being printed to the Shell area: you'll see that when the potentiometer is all the way one way, the numbers get very close to zero; when it's all the way the other way, they get very close to 3.3. These numbers represent the actual voltage being read by the pin – and as you turn the knob of the potentiometer, you're dividing the voltage smoothly between minimum and maximum, 0 V to 3.3 V.

Congratulations: you now know how to wire a potentiometer as both a varistor and a voltage divider, and how to read analogue inputs as both a raw value and a voltage!

Measuring temperatures

Your Raspberry Pi Pico's RP2040 microcontroller has an internal temperature sensor, which is read on the fourth analogue-to-digital converter channel. Like the potentiometer, the output of the sensor is a variable voltage: as the temperature changes, so does the voltage.

Start a new program, and import the machine and utime libraries:

```
import machine
import utime
```

Set up the analogue-to-digital converter again, but this time rather than using the number of a pin, use the channel number connected to the temperature sensor:

```
sensor_temp = machine.ADC(4)
```

You'll need your conversion factor again, to change the raw reading from the sensor into a voltage value, so add that:

```
conversion_factor = 3.3 / (65535)
```

Then set up a loop to take readings from the analogue input, apply the conversion factor, and store them in a variable:

```
while True:
    reading = sensor_temp.read_u16() * conversion_factor
```

Rather than print the reading directly, though, you need to do a second conversion – to take the voltage reported by the analogue-to-digital converter and convert it into degrees Celsius:

```
temperature = 27 - (reading - 0.706)/0.001721
```

This is another mathematical equation, and one which is specific to the temperature sensor in RP2040 . The values are taken from a technical document called a *data sheet* or *data book:* all electronic components have a data sheet, which is normally available on request from the manufacturer. You can view RP2040's data sheet in the Pico documentation at **rptl.io/rp2040-get-started** – it's packed full of information on how the microcontroller works, though it's aimed at engineers so is deeply technical.

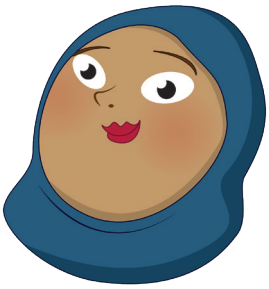
Finally, finish your loop:

```
print(temperature)
utime.sleep(2)
```

Your program will now look like this:

```
import machine
import utime

sensor_temp = machine.ADC(4)

conversion_factor = 3.3 / (65535)

while True:
    reading = sensor_temp.read_u16() * conversion_factor
    temperature = 27 - (reading - 0.706)/0.001721
    print(temperature)
    utime.sleep(2)
```

Click the Run icon and save your program as **Temperature.py**. Watch the Shell area: you'll see numbers being printed which represent the temperature reported by the sensor in degrees Celsius.



HEAT AND RP2040

If you have a traditional thermometer, you might see the figure reported by your Pico is a little higher: that's because the temperature sensor is located inside Pico's RP2040 chip, which is busily running your program. When the microcontroller is powered on, it's generating heat of its own – and that heat is enough to skew the result. For a simple program like this one, the skew might not be too high; if your program does a lot of complex calculations, the skew is likely to be higher.

Try gently pressing the tip of your finger to RP2040, the largest black chip in the middle of your Pico, and holding it there: the warmth of your finger should make the chip warmer, and the temperature will rise. Remove your finger from the chip, and the temperature will fall again.

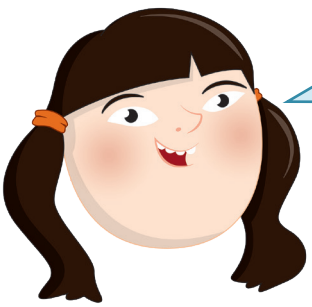
Congratulations – you've turned your Pico into a thermometer!

Fading an LED with PWM

The analogue-to-digital converter in your Pico only works one way: it takes an analogue signal and converts it to a digital signal the microcontroller can understand. If you want to go the other way, and have your digital microcontroller create an analogue output, you'd normally need a digital-to-analogue converter (DAC) – but there's a way to 'fake' an analogue signal, using something called *pulse-width modulation* or *PWM*.

A microcontroller's digital output can only ever be on or off, 0 or 1. Turning a digital output on and off is known as a *pulse* and by altering how quickly the pin turns on and off you can change, or *modulate*, the *width* of these pulses – hence 'pulse-width modulation'.

Every GPIO pin on your Pico is capable of pulse-width modulation, but the microcontroller's pulse-width modulation block is made up of eight slices, each with two outputs. Look at **Figure 8-4**: you'll see that each pin has a letter and a number. The number represents the PWM slice connected to that pin; the letter represents which output of the slice is used.



PWM CONFLICTS

You'll know if you accidentally use the same PWM output twice, because every time you alter the PWM values on one pin it will affect the conflicting pin as well. If that happens, take a look at the pinout diagram in **Figure 8-4** and your circuit and find a PWM output you haven't used yet.

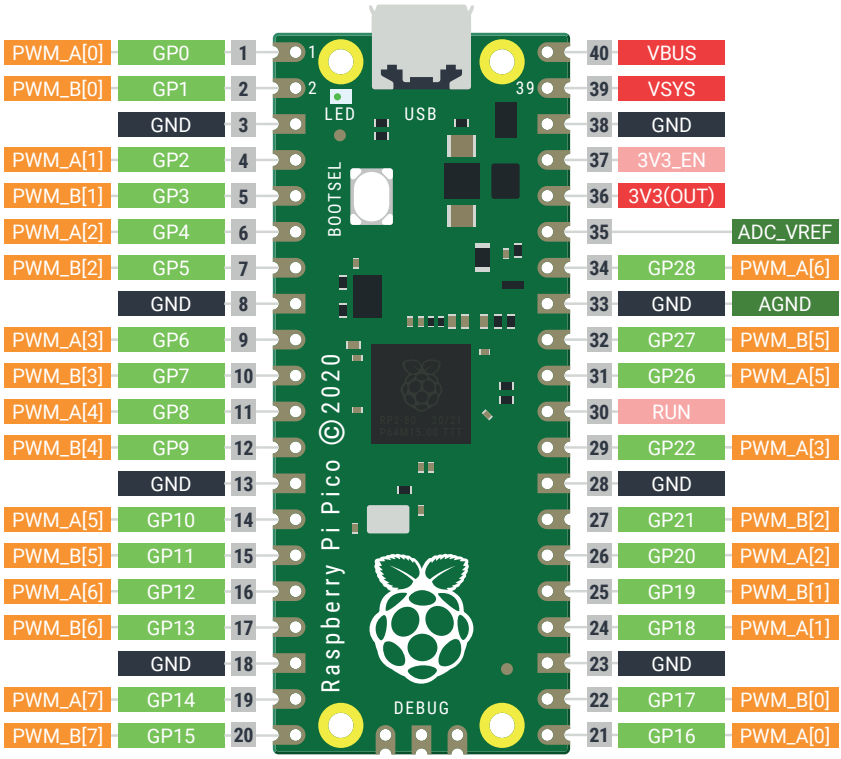


Figure 8-4: The pulse-width modulation pins

If that sounds confusing, don't worry: all it means is that you need to make sure you keep track of the PWM slices and outputs you're using, making sure to only connect to pins with a letter and number combination you haven't already used. If you're using PWM_A[0] on pin GP0 and PWM_B[0] on pin GP1, things will work fine, and will continue to work if you add PWM_A[1] on pin GP2; if you try to use the PWM channel on pin GP0 and pin GP16, though, you'd run into problems as they're both connected to PWM_A[0].

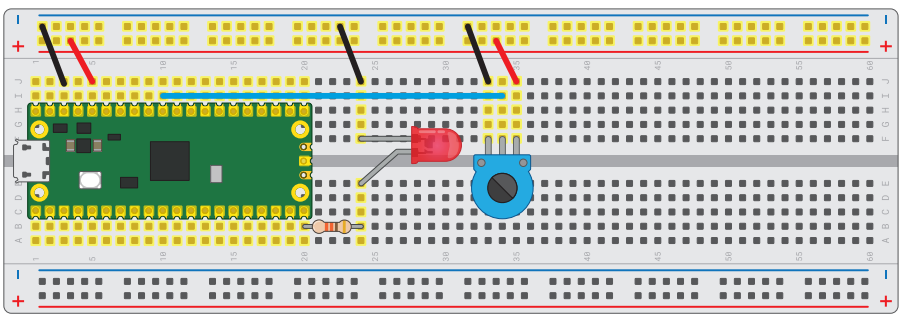


Figure 8-5: Adding an LED

Take an LED and a 330 Ω current-limiting resistor, and put them in the breadboard as shown in **Figure 8-5**. Wire the longer leg of the LED, the anode, to pin GP15 via the 330 Ω resistor, and wire the shorter leg to the ground pin of your Pico.

Go back to your first program by clicking on its tab just under Thonny's toolbar; if you'd already closed it, click the Open icon and load **Potentiometer.py** from your Pico. Just under where you set up the potentiometer as an analogue-to-digital input, type:

```
led = machine.PWM(machine.Pin(15))
```

This creates an LED object on pin GP15, but with a difference: it activates the pulse-width modulation output on the pin, channel B[7] – the second output of the eighth slice, counting from zero.

You'll also need to set the *frequency*, one of the two values you can change to control, or modulate, the pulse width. Add another line immediately below reading:

```
led.freq(1000)
```

This sets a frequency of 1000 hertz – one thousand cycles per second. Next, go to the bottom of your program and delete the **`print(voltage)`** and **`utime.sleep(2)`** lines before adding the following, remembering to keep it indented by four spaces so it forms part of the nested code within the loop:

```
led.duty(potentiometer.read_u16())
```

This line takes a raw reading from the analogue input connected to your potentiometer, then uses it as the second aspect of pulse-width modulation: the *duty cycle*. The duty cycle controls the pin's output: a 0 percent duty cycle leaves the pin switched off for all 1000 pulses per second, and effectively turns the pin off; a 100 percent duty cycle leaves the pin switched on for all 1000 pulses per second, and is functionally equivalent to just turning the pin on as a fixed digital output; a 50 percent duty cycle has the pin on for half the pulses and off for half the pulses.

Click Run and watch the LED as you turn the potentiometer: the LED will grow brighter with the potentiometer turned all the way one way, and gradually dimmer as you turn it the other. That's because the reading taken from the analogue pin connected to the potentiometer is being turned into a value for the PWM signal's duty cycle: a low duty cycle is like a low voltage on an analogue output, making the LED dim; a high duty cycle is like a high voltage, making the LED bright.

You'll notice, though, that the LED reaches its maximum brightness long before the potentiometer's knob has reached its stopping point, and very slight movements cause a large change in brightness. That's because the analogue reading is a number between 0 and 65,535, a 16-bit integer, but the duty cycle is at 0 percent with a setting of 0 and 100 percent with a value of just 1024. Anything above that value is simply ignored and treated as if it were 1024.

To fix that, and to make it so you can properly control the LED's brightness, you need to map the value from the analogue input to a range the PWM slice can understand. The best way to do this is to tell MicroPython that you're passing the duty cycle value as an unsigned 16-bit integer, the same number format as you receive from your Pico's analogue input pin.
Find the following line in your program:

```
led.duty(potentiometer.read_u16())
```

Edit it so it reads:

```
led.duty_u16(potentiometer.read_u16())
```

Your finished program will look like this:

```
import machine
import utime

potentiometer = machine.ADC(28)
led = machine.PWM(machine.Pin(15))
led.freq(1000)

while True:
    led.duty_u16(potentiometer.read_u16())
```

Click the Run icon and try turning the potentiometer all the way one way, then all the way the other. Watch the LED: this time, unless you're using a logarithmic potentiometer, you'll see the LED's brightness change smoothly from completely off at one end of the potentiometer knob's limit to fully lit at the other.

Congratulations: you've not only mastered analogue inputs, but you can now create the equivalent to an analogue output using pulse-width modulation!

CHALLENGE: CUSTOMISATION

Can you combine your two programs, and have the LED's brightness controlled by the temperature reading from the on-board temperature sensor? Can you remember how many analogue inputs your Pico has? What about PWM outputs? Try adding another analogue sensor to your Pico – something like a *light-dependent resistor (LDR)*, *gas sensor*, or *barometer* – and have your program read that instead of the potentiometer.

Chapter 9
Data logger

Turn Raspberry Pi Pico into a temperature data-logging device and untether it from the computer to make it fully portable

Throughout this book, you've been using your Raspberry Pi Pico connected to your Raspberry Pi or other computer via its micro USB port. As with all microcontrollers, though, there's no reason your Pico has to be tethered in this way: it's a fully functional self-contained system, with processing capabilities, memory, and everything it needs to work on its own.

In this chapter you'll learn how to use the file system to create, write to, and read from files, allowing you to put your Pico anywhere you like and have it record data for later access – turning it into what is known as a *data logger*. For this you'll only need your Pico and, if you want to use it away from your Raspberry Pi, a micro USB charger or battery pack; once you have finished the chapter, you can connect additional analogue sensors if you want to expand your project.

The file system

The file system is where your Pico stores all the programs you've been writing. It's equivalent in function to the microSD card in your Raspberry Pi, or the hard drive or solid-state drive in your laptop or desktop computer: it's a form of *non-volatile* storage, which means that whatever you save there stays in place even when you unplug your Pico's micro USB cable.

Connect your Pico to your Raspberry Pi and load Thonny, if you don't already have it open. Click the Open icon, and click MicroPython Device in the pop-up which appears. You'll see a list of all the programs you've been writing so far, stored on your Pico's file system. You're not going to open one right now, so click Cancel.

Click at the bottom of the Shell area to start working with your Pico in interactive mode. Type:

```
file = open("test.txt", "w")
```

This tells MicroPython to open a file called **test.txt** for writing – the **"w"** part of the instruction. You won't see anything print to the Shell area when you press **ENTER** at the end of the line, because although you've opened the file, you haven't done anything with it yet. Type:

```
file.write("Hello, File!")
```

When you press **ENTER** at the end of this line, you'll see the number 12 appear (**Figure 9-1**). That's MicroPython confirming to you that it has written 12 bytes to the file you opened. Count the number of characters in the message you wrote: including the letters, comma, space, and exclamation mark, there are twelve – each of which takes up a single byte.

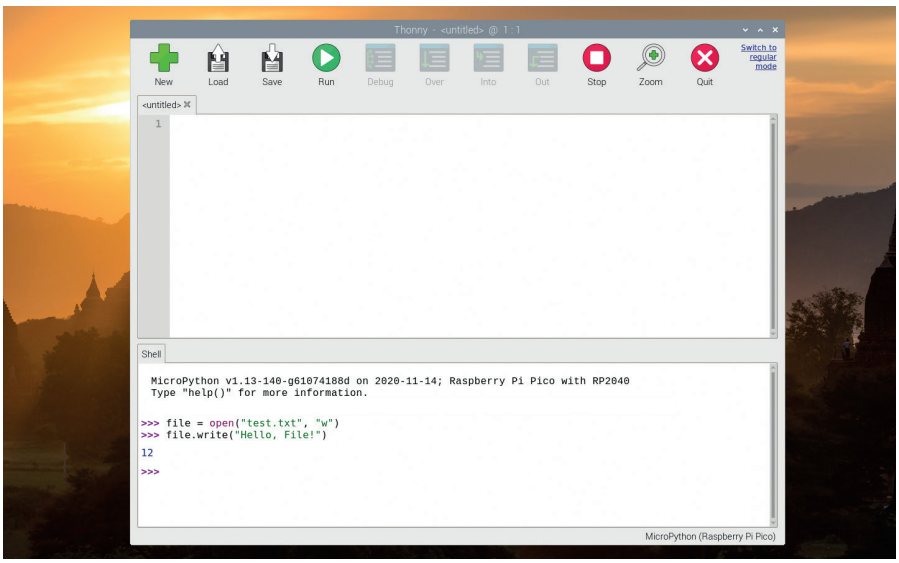


Figure 9-1: The size of the data you've written is printed to the Shell area

When you've written to a file, you need to close it – this ensures that the data you've told MicroPython to write is actually written to the file system. If you don't close the file, the data might not have been written yet – a bit like writing a letter in LibreOffice Writer or another word processor and forgetting to save it. Type:

```
file.close()
```

Your file is now safely stored on your Pico's file system. Click the Open icon on Thonny's toolbar, click MicroPython Device, and scroll through the list of files until you find **test.txt**. Click on it, then click OK to open it: you'll see your message pop up in a new Thonny tab.

You don't have to use the Open icon to read files, though: you can do it directly in MicroPython itself. Click back into the bottom of the Shell area and type:

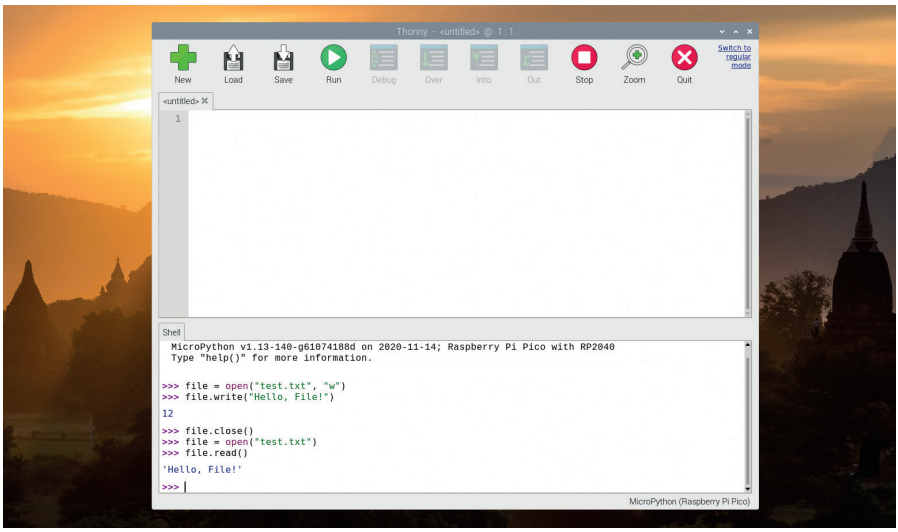
```
file = open("test.txt")
```

You'll notice that this time around there's no "`w`": that's because instead of writing to the file, you're going to be reading it. You could replace the "`w`" with an "`r`", but MicroPython defaults to opening a file in read mode – so it's fine to simply leave that part of the instruction off.

Next, type:

```
file.read()
```

You'll see the message you wrote to the file print to the Shell area (**Figure 9-2**). Congratulations: you can read and write files on your Pico's file system!



▲ **Figure 9-2:** Printing the stored message to the Shell area

Before you finish, remember to close the file – it's not as important to properly close a file after reading it as it is when writing to it, but it's a good habit to get into anyway:

```
file.close()
```

Logging temperatures

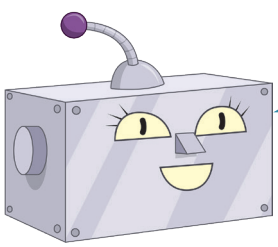
Now you know how to open, write to, and read from files, you have everything you need to build a data logger on your Pico. Click the New icon to start a new program in Thonny, and start your program by typing:

```
import machine
import utime

sensor_temp = machine.ADC(machine.ADC.CORE_TEMP)

conversion_factor = 3.3 / (65535)
reading = sensor_temp.read_u16() * conversion_factor
temperature = 27 - (reading - 0.706)/0.001721
```

You might recognise this code: it's the same as you used in **Chapter 8** to read from your Pico's on-board temperature sensor. The readings from the sensor are the data you're going to be logging to the file system, so you don't want to simply print them out as you did before.
Start by opening a file for writing by adding the following line at the bottom:

```
file = open("temps.txt", "w")
```

If the file doesn't already exist on the file system, this creates it; if it does, it overwrites it – emptying its contents ready for you to write new data.



WARNING

Opening a file for writing in MicroPython will delete anything you've already stored in it. Always make sure you've opened the file for reading and saved the contents somewhere if you want to keep it!

Now you need to write something to the file – the reading from the temperature sensor:

```
file.write(str(temperature))
```

Rather than writing a fixed string in quotes, as you did before, this time you're converting the variable temperature – which is a floating-point number, in other words one with a decimal point in it – to a string, then writing that to the file.

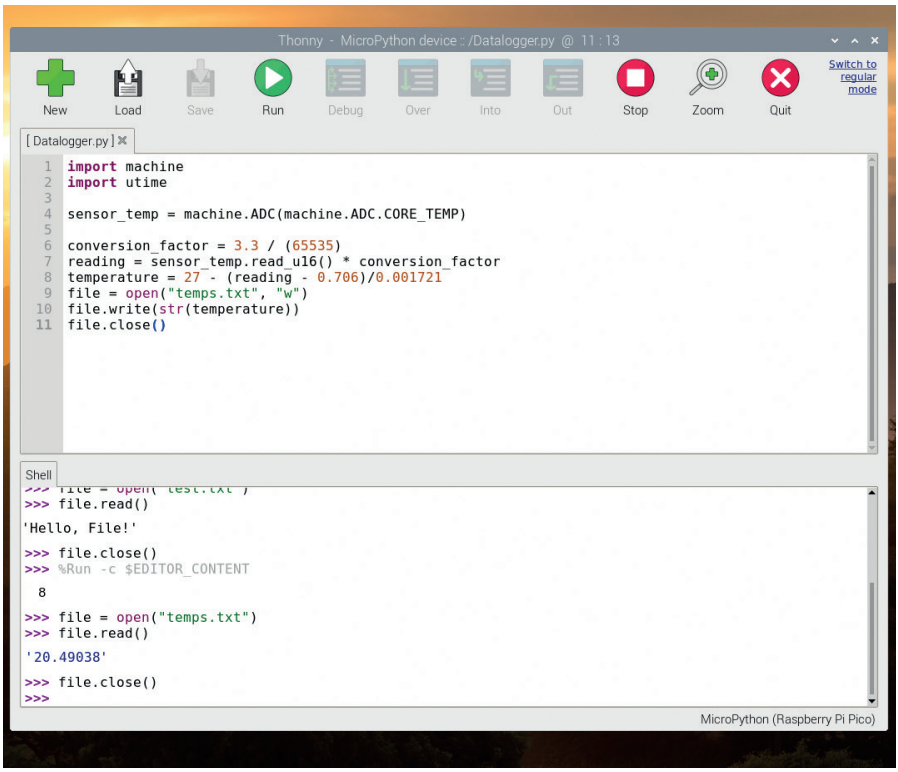
As before, to make sure the data is written you need to close the file:

```
file.close()
```

Click the Run icon and save your program to the MicroPython Device as **Datalogger.py.** The program will only take a few seconds to run; when the >>> prompt reappears at the bottom of the Shell area, click into it and type the following to open and read your new file:

```
file = open("temps.txt")
file.read()
file.close()
```

You'll see the temperature reading your program took appear in the Shell (**Figure 9-3**). Congratulations: your data logger works!



▲ **Figure 9-3:** Your file records the temperature at the time the measurement was taken

A data logger that only logs a single reading – a *datum* – isn't that useful, though. To make your data logger more powerful, you need to modify it so it takes lots of readings. Click Run again, and read the file again:.

```
file = open("temps.txt")
file.read()
file.close()
```

Notice how there's still only one reading in the file. When your program opened the file for writing again, it automatically wiped its previous contents – meaning that each time your program runs, it will wipe the file and store a single reading.

To fix that, you need to modify your program. Start by clicking and dragging your mouse cursor to highlight the lines:

```
reading = sensor_temp.read_u16() * conversion_factor
temperature = 27 - (reading - 0.706)/0.001721
```

When you've highlighted both lines – making sure not to miss any parts – let go of the mouse button and press **CTRL+X** on your keyboard to *cut* the lines; you'll see them disappear from the program.

Next, go to the bottom of your program and delete everything after:

```
file = open("temps.txt", "w")
```

Now type:

```
while True:
```

After pressing **ENTER** at the end of that line, hold down the **CTRL** key and press the **V** key to *paste* the two lines you cut earlier. You'll see them appear, which saves you having to type them in – but only the first line will be properly indented so it's nested as part of the infinite loop you just opened. Put your cursor at the start of the line below by clicking and press the **SPACE** bar four times to indent the line properly, then move your cursor to the end of the line and press **ENTER.**

Type the following line, making sure it's properly indented:

```
file.write(str(temperature)
```

Now, though, you're going to need to do something new. If you close the file as you did before, you won't be able to write to it again without reopening it and wiping its contents. If you don't close the file, the data will never actually get written to the file system.

The solution: *flush* the file, rather than close it. Type:

```
file.flush()
```

When you're writing to a file but the data isn't actually being written to the file system, it's stored in what's known as a *buffer* – a temporary storage area. When you close the file, the buffer is written to the file in a process known as *flushing*. Using **`file.flush()`** is equivalent to **`file.close()`**, in that it flushes the contents of the buffer into the file – but unlike **`file.close()`**, the file remains open for you to write more data to it later.

Now you just need to pause your program between readings:

```
utime.sleep(10)
```

Your finished program will look like this:

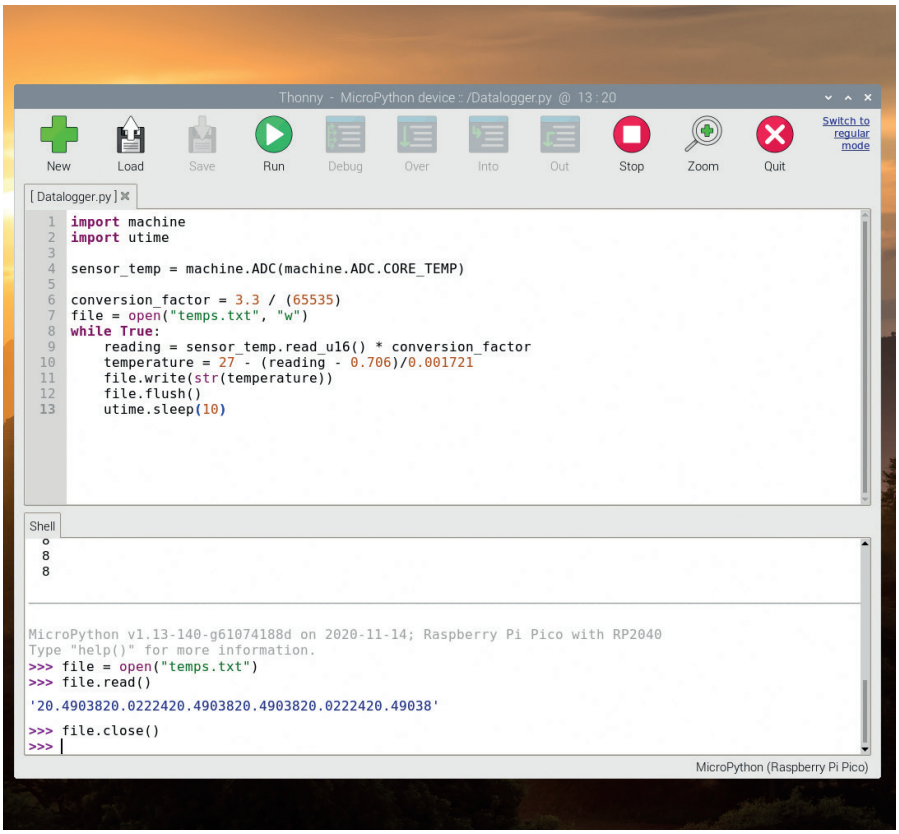
```
import machine
import utime

sensor_temp = machine.ADC(machine.ADC.CORE_TEMP)

conversion_factor = 3.3 / (65535)
file = open("temps.txt", "w")
while True:
    reading = sensor_temp.read_u16() * conversion_factor
    temperature = 27 - (reading - 0.706)/0.001721
    file.write(str(temperature))
    file.flush()
    utime.sleep(10)
```

Click the Run icon and count to 60, then click the Stop icon. Now open and read your file in the script area:

```
file = open("temps.txt")
file.read()
file.close()
```

The good news is that your program has worked, and you've recorded multiple readings in a single file – around six, give or take a few depending on how quickly you counted. The bad news is that they're all mushed together on a single line (**Figure 9-4**) – making it difficult to read.

To fix that problem, you need to format the data as it's written to the file. Go back to the **`file.write()`** line in your program, and modify it so it looks like:



▲ **Figure 9-4:** All the measurements are there, but the formatting makes it difficult to read

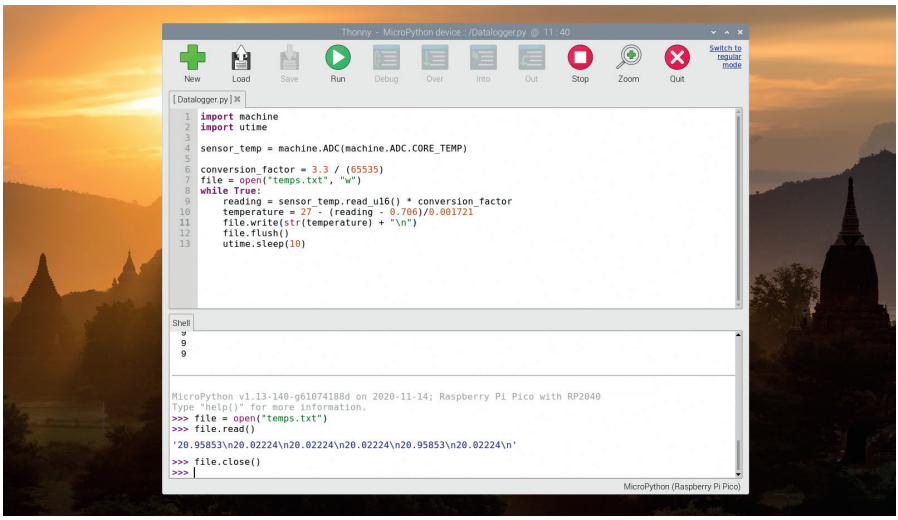
```
file.write(str(temperature) + "\n")
```

The plus symbol (+) tells MicroPython that you want to append what follows, concatenating the two strings together; **"\n"** is a special string known as a *control character* – it acts as the equivalent of pressing the **ENTER** key, meaning that each line in your data log should be on its own separate line.

Click the Run icon, count to 60 again, and click Stop. Open and read your file:

```
file = open("temps.txt")
file.read()
file.close()
```

You've made progress, but it's still not right: the **\n** control character isn't acting like a press of **ENTER**, but printing as two visible characters (**Figure 9-5**, overleaf). That's because **file.read()** is bringing in the raw contents of the file, and making no attempt at formatting it for the screen.

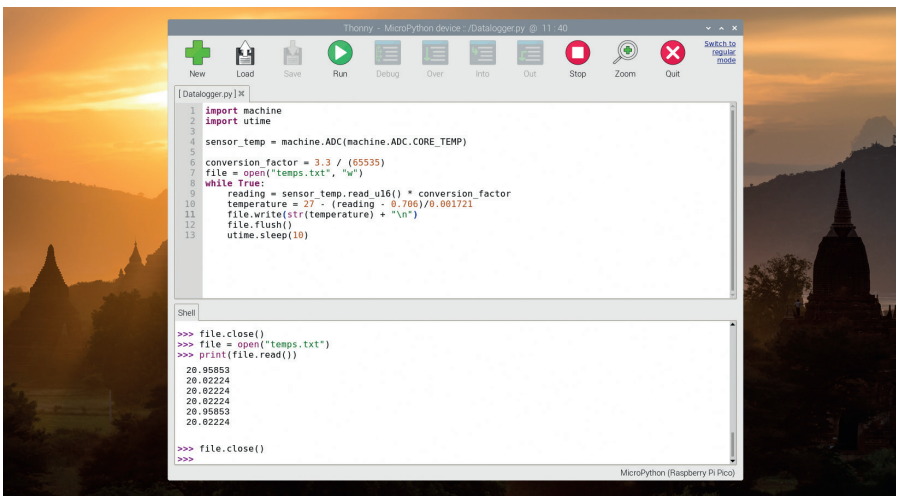


▲ **Figure 9-5:** You're getting closer to an easy-to-read printout here

To fix the formatting problem, you need to wrap the file read in a **`print()`** function:

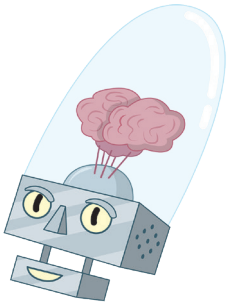
```
file = open("temps.txt")
print(file.read())
file.close()
```

This time you'll see each reading print out on its own line, neatly formatted and easy to read (**Figure 9-6**).



▲ **Figure 9-6:** Now you can read all the temperatures your data logger has captured

Congratulations: you've built a data logger which can take multiple readings and store them on your Pico's file system!



FILE STORAGE

Your Pico's file system is 1.375MiB in size, meaning it can hold 1,441,792 bytes of data. Every file you save on your Pico, including the data logger's storage file, takes up room. How long it takes to fill the storage will depend on how many other files you have and how often your data logger saves a reading: at nine bytes per reading every ten seconds, you'll fill 1.375MiB in around 18.5 days; if you took a reading every minute instead, your data logger could run for around 111 days; if you only read once an hour, your data logger could run for more than 18 years!

Your Pico's file system works regardless of whether or not it's connected to your Raspberry Pi or another computer. If you have a micro USB mains charger or a USB battery pack with a micro USB cable, you can take your data logger to any room in your house and have it run by itself – but you'll need a way to get your program running without having to click the Run icon in Thonny.

For use without a connected computer – known as *headless operation* – you can save your program under a special file name: **main.py**. When MicroPython finds a file called **main.py** in its file system, it runs that automatically every time it's powered on or reset – without you having to click Run.

In Thonny, after stopping the program if running, click the File menu then Save As. Click MicroPython Device in the pop-up that appears, then type 'main.py' as the file name before clicking Save. At first, nothing will seem to happen: that's because Thonny puts your Pico into interactive mode, which stops it from automatically running the program you just saved.

To force the program to run, click into the bottom of the Shell area and hold down the **CTRL** key, press the **D** key, then let go of the **CTRL** key. This sends your Pico a *soft reset* command, which will break it out of interactive mode and start the program running. Find something else to do for five minutes or so, then press the Stop icon and open your data log:

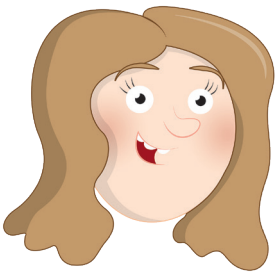
```
file = open("temps.txt")
print(file.read())
file.close()
```

You'll see a list of temperature readings, even though you didn't click the Run icon – because your program ran automatically when your Pico reset.

If you have a micro USB charger or USB battery pack, disconnect your Pico from your Raspberry Pi, take it to another room, and connect it to the charger or battery pack. Leave it there

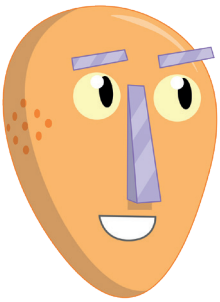
for ten minutes, then come back and unplug it. Take it back to your Raspberry Pi, plug it back in, and read your file again: you'll see the readings from the other room, proving that your Pico can run perfectly well without your Raspberry Pi helping it along.

Congratulations: your data logger is now fully functional and wholly portable, ready to go with you wherever you need to record data!



WARNING

Your data logger program will run every time your Pico is powered on without being connected to Thonny. If you don't want that to happen, you can simply open **main.py** in Thonny and delete all the code in it before saving it again. With an empty **main.py**, your Pico will simply sit and wait for instructions again.



CHALLENGE

Can you change your program to record data from an external sensor connected to one of your Pico's ADC pins? Can you have your program write a title at the start of the file, so it's easier to see what the values mean? Can you write a program which logs how many times a push-button switch has been pressed? Can you figure out a way, such as copy-and-paste, to get your data into LibreOffice Calc or another spreadsheet program to create a chart?

Chapter 10

Digital communication protocols: I2C and SPI

Explore these two popular communication protocols and use them to display data on an LCD

So far in this book we've looked at how to work with a few common bits of hardware, but as you build more projects on your own, you'll probably want to branch out to use all sorts of different sensors, actuators, and displays. How will you communicate with these? Sometimes you may find that there's a MicroPython library you can use where someone's already converted the low-level functions into an easy-to-use interface. However, this isn't always the case.

Fortunately, there are a couple of standard ways of connecting low-level digital devices together that are implemented in MicroPython: Inter-Integrated Circuit (I2C) and Serial Peripheral Interface (SPI). In many ways, they're very similar in that they both define a way of wiring up

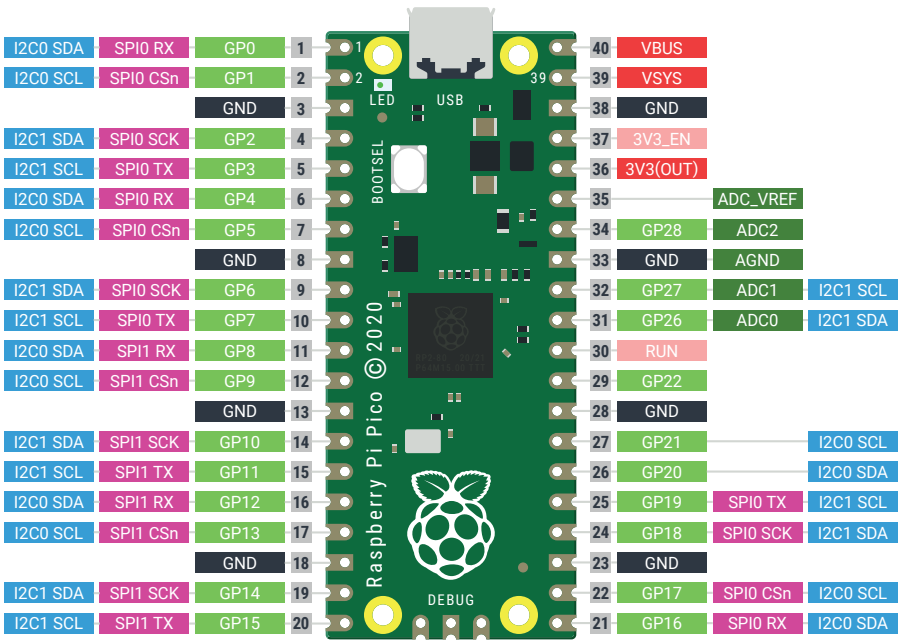
a two-way interface between two devices. In fact, many parts come in versions with either interface, so you can pick the one that's right for your project. In both cases, there's one device that controls the communication (your Pico) and one (or more) that waits for instructions from the main device. However, there are a few differences. We'll look at both protocols now and then we'll help you pick the right one for each project.

VOLTAGE LEVELS

Your Pico's GPIO pins work at 3.3 volts. Applying a higher voltage to them may damage them. Fortunately, this is a common voltage to work at and a large proportion of the devices you come across will work at 3.3 volts. However, before plugging some new hardware into your Pico, always double-check that it's a 3.3 V device, as both I2C and SPI devices can run at 5 V sometimes and this will damage your Pico.

I2C

Communication over I2C takes place on two wires: a clock (usually marked as SCL) and a data channel (usually marked SDA).



▲ **Figure 10-1:** The pinout for Raspberry Pi Pico. I2C functions are shown in light blue; SPI in pink

These have to connect to specific pins on the Pico. There are a few choices; take a look at the pinout diagram for the options (**Figure 10-1**). There are two I2C buses (I2C0 and I2C1), and you can use either or both. In our example, we'll use I2C0 – with GP0 for SDA, and GP1 for SCL.

To demonstrate the protocols, we'll use a SerLCD module from SparkFun. This has the advantage that it has both I2C and SPI interfaces, so we can see the differences between the two methods with the same hardware.

This LCD can display two lines, each with up to 16 characters. It's a useful device for outputting bits of information about our system. Let's take a look at how to use it.

Wiring I2C is just a case of connecting the SDA pin on the Pico with the SDA pin on the LCD and the same for the SCL. Because of the way I2C handles communication, there also needs to be a resistor connecting SDA to 3.3 V and SCL to 3.3 V. Typically these are about 4.7 kΩ. However, with our device, these resistors are already included, so we don't need to add any extra ones.

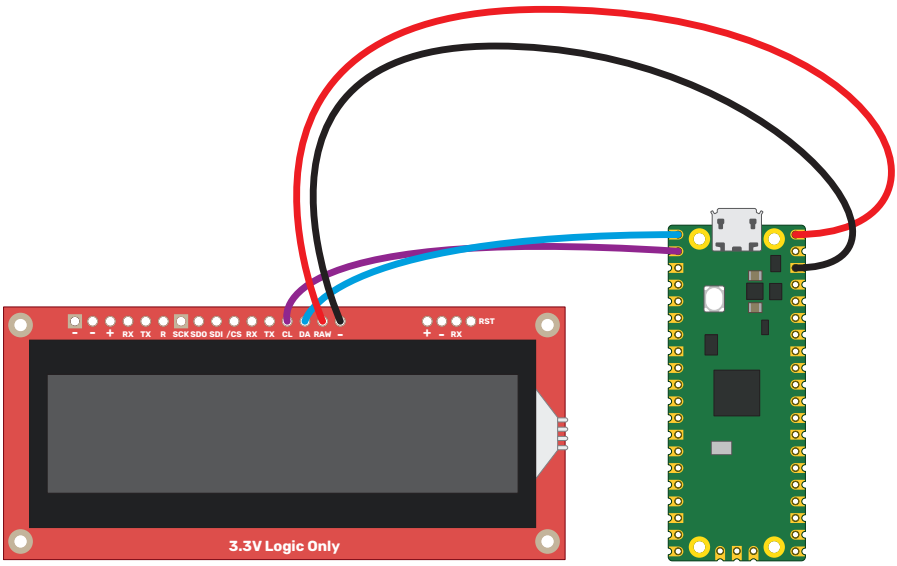


Figure 10-2: Wiring up a SerLCD module for I2C

With this wired up (see **Figure 10-2**), displaying information on the screen is as simple as:

```
import machine
sda=machine.Pin(0)
scl=machine.Pin(1)
i2c=machine.I2C(0,sda=sda, scl=scl, freq=400000)
i2c.writeto(114, '\x7C')
i2c.writeto(114, '\x2D')
i2c.writeto(114, "hello world")
```

This code doesn't do very much. It connects to the I2C device and sends some data. However, there are a few bits that may look a little unusual.

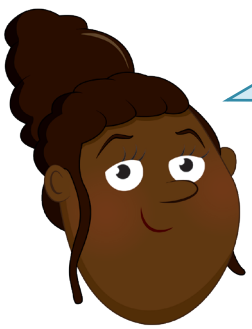
The 114 in the **i2c.writeto()** lines refers to the address of the I2C device. You can connect many devices to an I2C bus (more on this later), and each time you want to send or receive data, you need to specify the address of the device you want to communicate with. This address is hard-wired into the device (though you may be able to change it by cutting a trace on the PCB, or soldering a blob – see your device's documentation for details).

You should find the address for your device in the documentation, but you can scan an I2C bus to see what addresses are currently in use. After setting up the I2C bus, you can run the **scan** method to output the addresses currently in use:

```
import machine
sda=machine.Pin(0)
scl=machine.Pin(1)
i2c=machine.I2C(0,sda=sda, scl=scl, freq=400000)
print(i2c.scan())
```

The next bits that may look a little odd are the **\x7C** and **\x2D** commands that are written. Each I2C device requires data sent in a specific format. There's no standard for this, so you'll have to refer to the documentation for whatever I2C device you're setting up. The **\x** at the start of each of these tells MicroPython that we're sending a hexadecimal string (see 'Hexadecimal' box) which is a common way of ensuring you're sending the exact data you want. For our LCD, 7C enters command mode and 2D blanks the LCD and sets the cursor to the beginning. Following this, we can send data that's displayed on the screen:

```
file.close()
```



HEXADECIMAL

Hexadecimal is a base-16 numbering system. That means there are 16 digits: 0–F. So, the number 10 in decimal is A in hexadecimal. 17 in decimal is 11 in hexadecimal. The advantage of this is that each digit contains two bytes' worth of information. This makes it a compact but still understandable way of writing digital information. You'll come across it quite a lot when dealing with devices that take their instructions as digital values (such as our LCD).

If you get confused, you can use online hexadecimal-to-decimal converters to switch between the two. For example: **hsmag.cc/hextodec.**

Of course, there's not much use in a screen that just says Hello World, so let's take a look at turning this into something a little more useful – a thermometer. In **Chapter 8** you learned how to use the ADC to read temperatures using your Pico's internal temperature sensor. We can now build on this code to make a standalone thermometer that doesn't need a computer to read the output. With your LCD still connected as before, run the following code:

```
import machine
import utime

sda=machine.Pin(0)
scl=machine.Pin(1)
i2c=machine.I2C(0,sda=sda, scl=scl, freq=400000)

adc = machine.ADC(4)
conversion_factor = 3.3 / (65535)
while True:
    reading = adc.read_u16() * conversion_factor
    temperature = 25 - (reading - 0.706)/0.001721
    i2c.writeto(114, '\x7C')
    i2c.writeto(114, '\x2D')
    out_string = "Temp: " + str(temperature)
    i2c.writeto(114, out_string)
    utime.sleep(2)
```

This should mostly look familiar. The only slight change to the previous temperature code is that before we outputted the result of our calculation – a number – but the LCD needs characters to display, so we use the **`str`** function which converts the number to a string of characters. We can then build this into a slightly more informative output by combining it with **`"Temp: "`**.

As you've seen, I2C is an easy way of linking extra hardware to your Pico. You will need to ensure you've got appropriate documentation for whatever device you want to connect that lets you know what commands do what, but as long as you know this, you can easily add all sorts of bits and bobs to your Pico and create impressive builds.

Serial Peripheral Interface

We've seen how I2C works, now let's take a look at SPI. We'll use the exact same LCD, so the commands and everything else are the same, it's just the protocol we send data over that's different.

SPI has four connections: SCLK, MOSI, MISO, and CS (sometimes labelled SS). SCLK is the clock, MOSI is the line taking data from your Pico to the peripheral device (see 'SPI terminology' box), and MISO takes data from your peripheral device to your Pico. CS stands for Chip Select and is used to connect many devices to a single SPI bus. You simply have to apply power to the

CS line to enable an SPI peripheral and pull it low to disable it. To confuse things slightly, this particular device doesn't have CS, but /CS which stands for NOT CS – in other words, it's the opposite of CS, so you bring it low to enable the LCD and high to disable it. You could connect the CS to a GPIO pin and toggle this on and off to enable and disable the display, but since we only have one device, we can simply connect it to ground to keep it enabled (**Figure 10-3**).

So, with the SerLCD's power lines connected to VBUS and GND, we just need to connect its SDO to Pico's MISO (GP4 / SPI0 RX), SDI to MOSI (GP3 / SPI0 TX), SCK to SCLK (GP2 / SPI0 SCK), and /CS to GND.

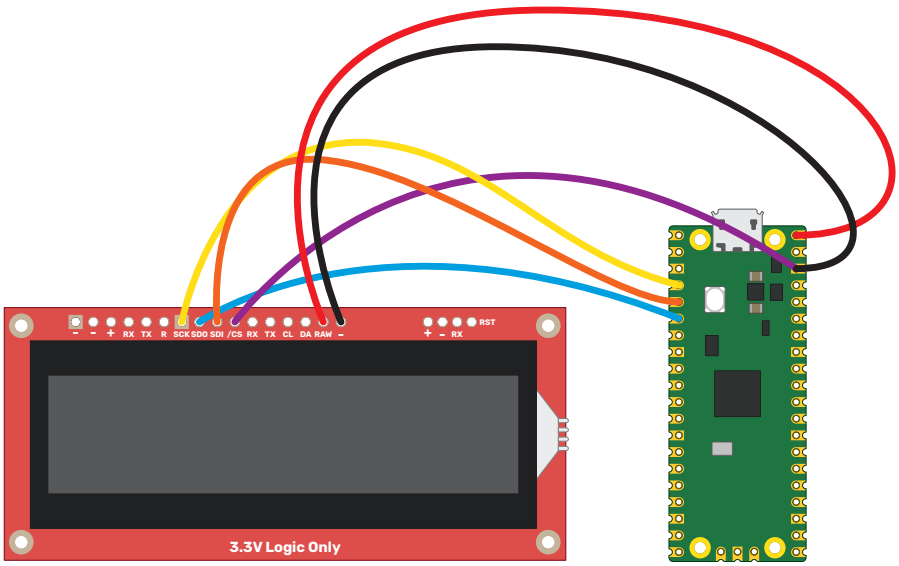
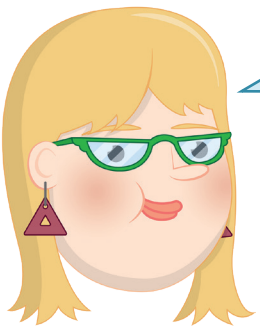


Figure 10-3: Wiring up a SerLCD module for SPI



SPI TERMINOLOGY

SPI requires four connections: one that takes data from the master device to the slave device, another that takes data in the opposite direction, plus power and ground. Two data wires mean that data can travel in both directions at the same time. These are usually called Master Out Slave In (MOSI) and Master In Slave Out (MISO). However, you will come across them with different names. If you look at the Raspberry Pi Pico pinout (Appendix B), they're referred to as SPI TX (Transmit) and SPI RX (Receive). This is because Pico can be either a master or slave device, so whether these connections are MOSI or MISO depends on the current function of Pico. On the LCD we're using, they're labelled SDI (Serial Data In) and SDO (Serial Data Out).

There are no addresses in SPI, so we can just dive in and write our code:

```
import machine

spi_sck=machine.Pin(2)
spi_tx=machine.Pin(3)
spi_rx=machine.Pin(4)

spi=machine.SPI(0,baudrate=100000,sck=spi_sck, mosi=spi_tx, miso=spi_rx)
spi.write('\x7C')
spi.write('\x2D')
spi.write("hello world")
```

In this case, we're using SPI0, and one set of available pins for this is GP2, GP3, and GP4. Most types of serial communication have a speed or baudrate, which is basically how many bits of data it can push through the channel per second. A lot of things affect this, such as the capabilities of the two devices being connected and the wiring between them (how long it is and if there's interference from other devices). If you find you're having problems with mangled data, then you may need to reduce it. For our little screen, we're just sending one byte of data per character, so it doesn't really matter how fast we send it, but for some other SPI devices (such as pixel-based displays), fine-tuning the baud rate can be important.

Let's have take a look at how this leaves our thermometer code:

```
import machine
import utime

spi_sck=machine.Pin(2)
spi_tx=machine.Pin(3)
spi_rx=machine.Pin(4)

spi=machine.SPI(0,baudrate=100000,sck=spi_sck, mosi=spi_tx, miso=spi_rx)

adc = machine.ADC(4)
conversion_factor = 3.3 / (65535)

while True:
    reading = adc.read_u16() * conversion_factor
    temperature = 25 - (reading - 0.706)/0.001721
    spi.write('\x7C')
    spi.write('\x2D')
    out_string = "Temp: " + str(temperature)
```

```
    spi.write(out_string)
    utime.sleep(2)
```

As you can see, there's really very little difference in the code between I2C and SPI. Once you've got everything set up, the only really change is that with I2C you have to specify the address when you send data, while with SPI you don't (though remember if you had more than one device attached, you'd need to toggle the CS GPIO to select the appropriate device).

So, if they're so similar, which protocol should you choose when building a project? There are a few factors to consider. The first is availability of the things you want to attach. Sometimes a sensor is only available as I2C or SPI, so you have to use that. However, if you've got a choice of hardware, the biggest impact comes when you're using multiple extra devices. With I2C, you can connect up to 128 devices to a single I2C bus; however, they all need to have a separate address. These addresses are hard-wired in. Sometimes it's possible to change the address with a solderable (or cuttable) connection, but sometimes it's not. If you want to have multiple of the same type of sensors (for example, if you're monitoring the temperature at many points on your project), you may be limited by the number of I2C addresses for your sensor. In this case, SPI may be a better choice.

Alternatively, SPI can have an unlimited number of devices connected; however, each one has to have its own CS line. On the Pico, there are 26 GPIO pins. You need three of them for the SPI bus, so that means there are 23 available for CS lines. And this is assuming you don't need any for anything else. If available GPIOs are at a premium, then you may want to look at I2C.

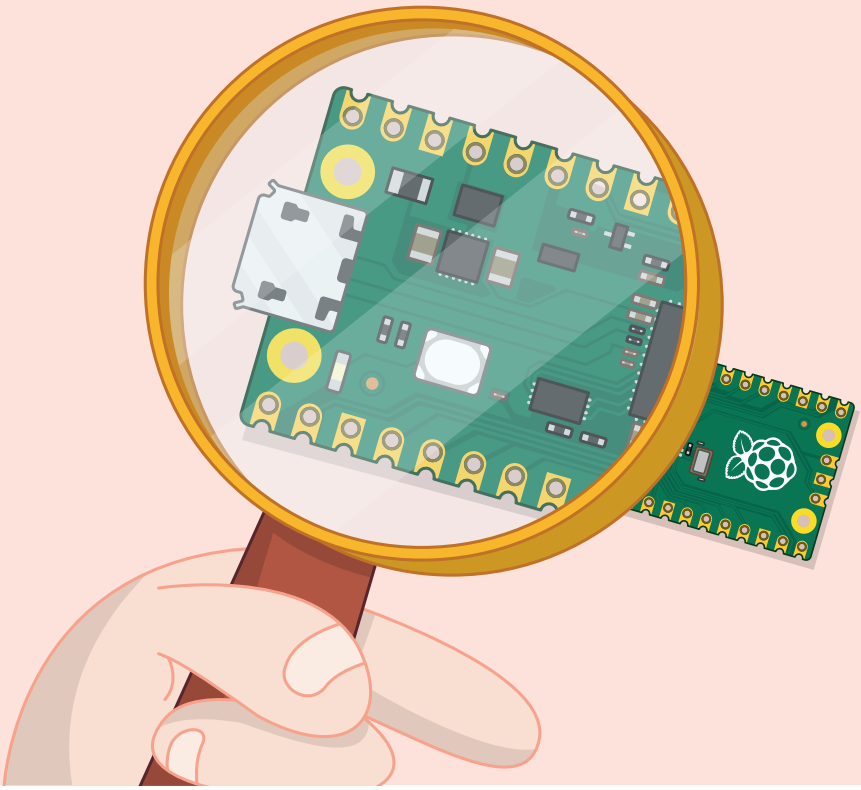
In reality, for many projects, you can quite happily use either protocol, and you may find that the choice of which to use has more to do with what parts you find in your parts box than a technical difference between the two.

BIT BANGING

Your Pico has two hardware I2C buses and two hardware SPI buses. However, you can use more than these if you want to. Both I2C and SPI can be implemented in software rather than hardware. This means the main processing core handles the communication protocol rather than a specialised bit of the microcontroller. This is known as 'bit banging'. While it can be useful, it puts more strain on your processor core than using specialised hardware, and you may find that you can't reach high baudrates.

The Pico has a trick up its sleeve for this – PIO. We'll look more closely at this later in the book (**Appendix C**), but it's an extra bit of hardware in the microcontroller that can be dedicated to input/output protocols such as I2C and SPI. With PIO, you can create extra I2C or SPI buses without taxing the main processor core.

Appendix A
Raspberry Pi Pico specifications



The various components and features of a microcontroller are known as its *specifications*, and a look at the specifications gives you the information you need to compare two microcontrollers.

These specifications can seem confusing at first, are highly technical, and you don't need to know them to use your Raspberry Pi Pico, but they are included here for the curious reader.

Raspberry Pi Pico's microcontroller chip is a Raspberry Pi RP2040, which you'll see indicated by markings etched into the top of the component if you look closely enough. The microcontroller's name can be broken down into sections, each of which has a particular meaning:

- **RP** means 'Raspberry Pi', simply enough.

- **2** is the number of *processor cores* the microcontroller has.

- **0** is the type of processor core, indicating in this case the RP2040 uses a processor core called the *Cortex-M0+* from Cambridge-based Arm.

- **4** is how much *random access memory (RAM*) the microcontroller has, based on a special mathematical function: *floor(log_2(RAM/16))*. In this case, '4' means the chip has *264 kilobytes (kB)* of RAM.

- **0** is how much *non-volatile (NV) storage* the chip has, and is worked out in the same way as the RAM: *floor(log_2(NV/16))*. In this case, 0 simply means there is no non-volatile storage on-board.

The RP2040 is the first microcontroller from Raspberry Pi; when future models are released, these numbers will be used so you can quickly see how their features compare at a glance.

Your Pico's two Cortex-M0+ processor cores run at 48MHz (48 million cycles per second), though this can be changed in software up to 133MHz (133 million cycles per second) if your program needs higher performance.

The microcontroller's RAM is built into the same chip as the processor's cores themselves, and takes the form of six individual memory banks totalling 264kB (264,000 bytes) of static RAM (SRAM). The RAM is used to store your programs and the data they need.

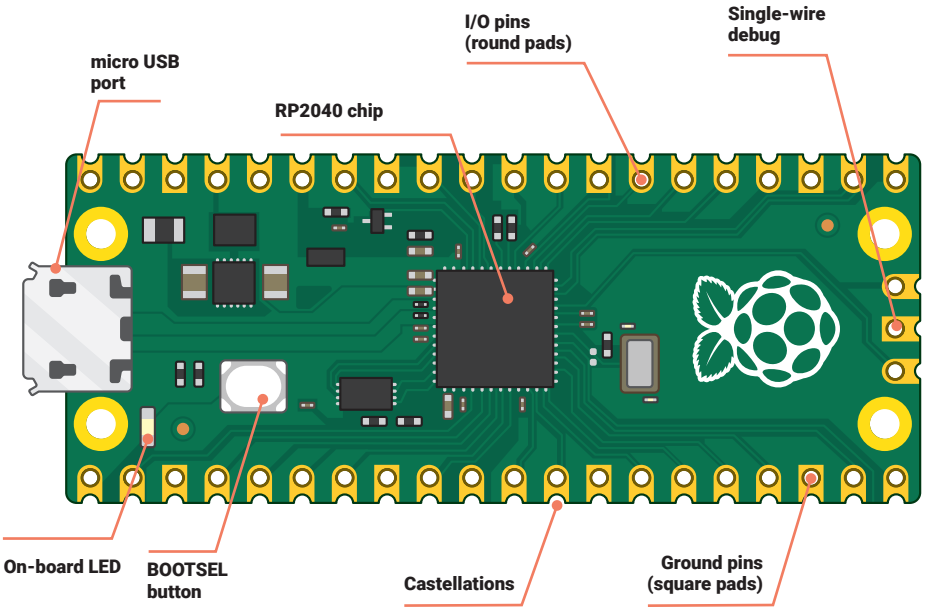
RP2040 includes 30 multifunction general-purpose input/output (GPIO) pins, 26 of which are brought out to physical pin connectors on your Pico and one of which is connected to an on-board LED. Three of these GPIO pins are connected to an analogue-to-digital converter (ADC), while another ADC channel is connected to an on-chip temperature sensor.

RP2040 includes two *universal asynchronous receiver-transmitter (UART)*, two *serial peripheral interface (SPI)*, and *two inter-integrated circuit (I2C) buses* for connections to external hardware devices like sensors, displays, digital-to-analogue converters (DACs), and more. The microcontroller also includes *programmable input/output (PIO),* which lets the programmer define new hardware functions and buses in software.

Your Pico includes a micro USB connector, which provides a UART-over-USB serial link to the RP2040 microcontroller for programming and interaction, and which powers the chip. Holding down the BOOTSEL button when plugging the cable in will switch the microcontroller into *USB Mass Storage Device mode*, allowing you to load new *firmware.*

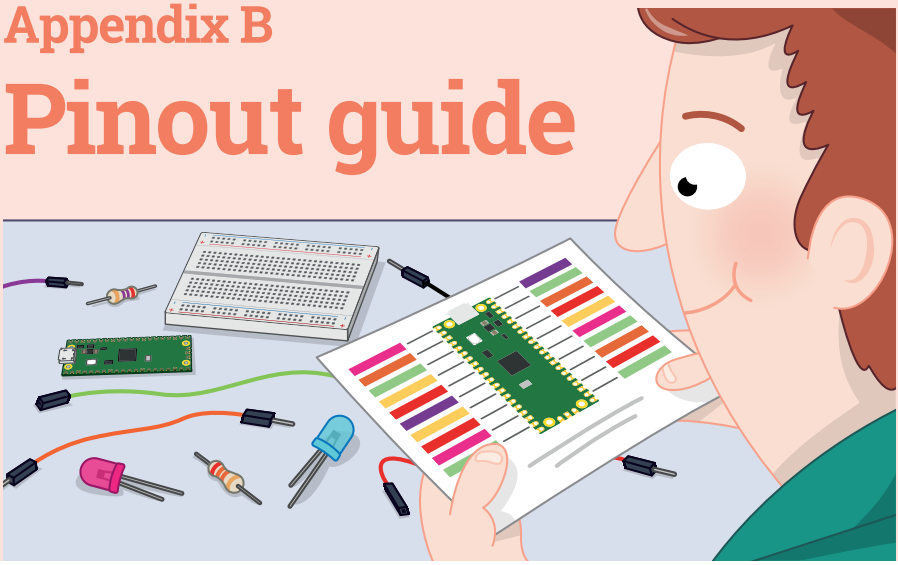
RP2040 also includes an accurate *on-chip clock and timer,* which allows it to keep track of the time and date. The clock can store the year, month, day, day of the week, hour, minute, and second, and automatically keeps track of elapsed time as long as power is provided.

Finally, RP2040 includes *single-wire debug (SWD)* for hardware debugging purposes, brought out to three pins at the bottom of your Pico.



- **CPU:** 32-bit dual-core ARM Cortex-M0+ at 48MHz, configurable up to 133MHz
- **RAM:** 264kB of SRAM in six independently configurable banks
- **Storage:** 2MB external flash RAM
- **GPIO:** 26 pins
- **ADC:** 3 × 12-bit ADC pins
- **PWM:** Eight slices, two outputs per slice for 16 total
- **Clock:** Accurate on-chip clock and timer with year, month, day, day-of-week, hour, second, and automatic leap-year calculation
- **Sensors:** On-chip temperature sensor connected to 12-bit ADC channel)
- **LEDs:** On-board user-addressable LED
- **Bus Connectivity:** 2 × UART, 2 × SPI, 2 × I2C, Programmable Input/Output (PIO)
- **Hardware Debug:** Single-Wire Debug (SWD)
- **Mount Options:** Through-hole and castellated pins (unpopulated) with 4 × mounting holes
- **Power:** 5 V via micro USB connector, 3.3 V via 3V3 pin, or 2–5V via VSYS pin

Appendix B
Pinout guide



Raspberry Pi Pico exposes 26 of the 30 possible RP2040 GPIO (general-purpose input/output) pins by routing them straight out to Pico header pins. GP0 to GP22 are digital only and GP 26–28 are able to be used either as digital GPIO or as ADC (analogue-to-digital converter) inputs, selectable in software. Most of the GPIO pins also offer secondary functionality for SPI, I2C, or UART communication protocols. All GPIO pins may also be used with PWM (pulse-width modulation) – see **Chapter 8** for more details.

VBUS is the micro-USB input voltage, connected to micro-USB port pin 1. This is nominally 5 V (or 0 V if the USB is not connected or not powered).

VSYS is the main system input voltage, which can vary in the allowed range 1.8 V to 5.5 V, and which is used by the on-board SMPS (switch mode power supply) to generate the 3.3 V for the RP2040 and its GPIO.

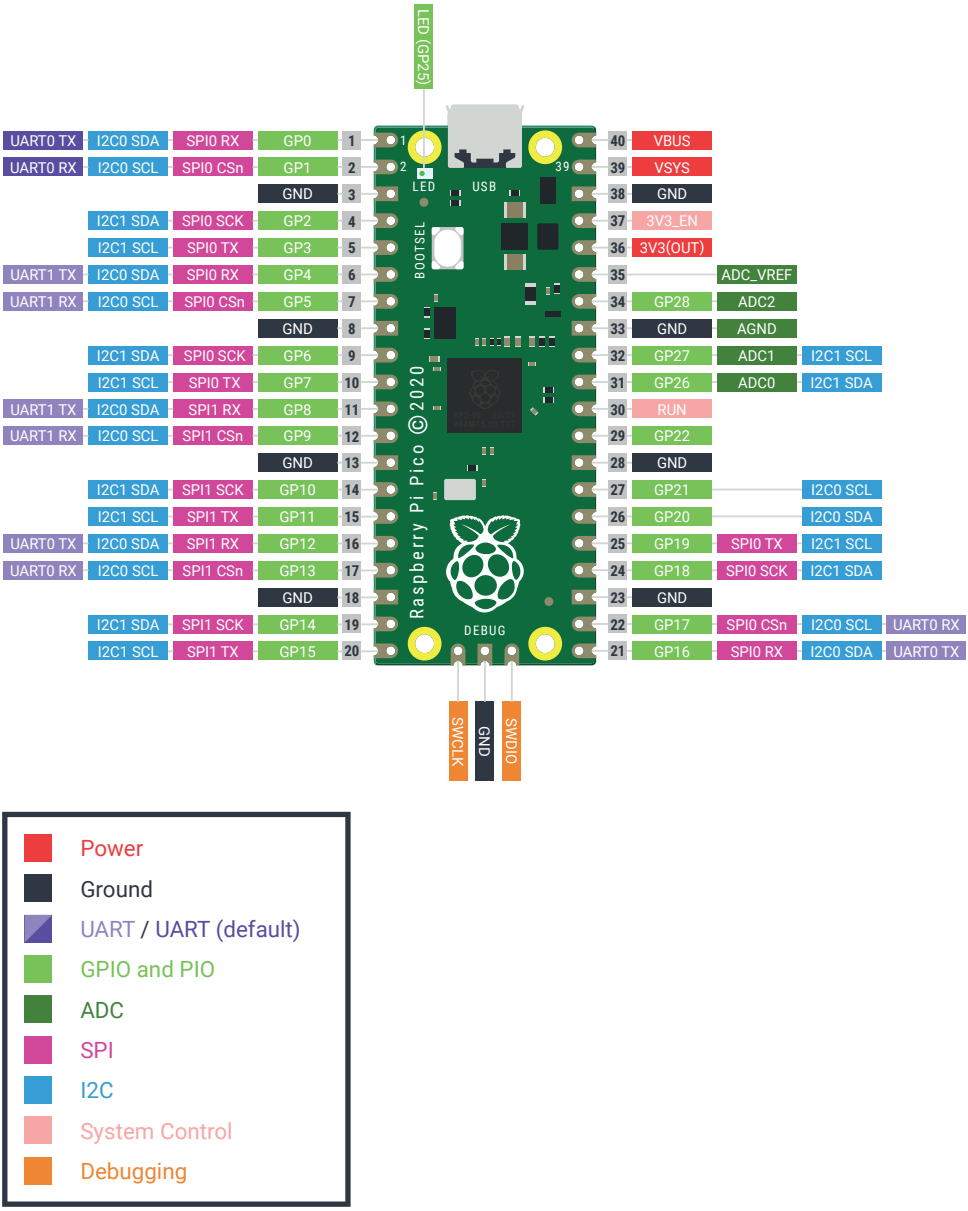
3V3_EN connects to the on-board SMPS enable pin, and is pulled high (to VSYS) via a 100 kΩ resistor. To disable the 3.3 V (which also de-powers the RP2040), short this pin low.

3V3 is the main 3.3 V supply to RP2040 and its IO, generated by the on-board SMPS. This pin can be used to power external circuitry (maximum output current will depend on RP2040 load and VSYS voltage, it is recommended to keep the load on this pin less than 300 mA).

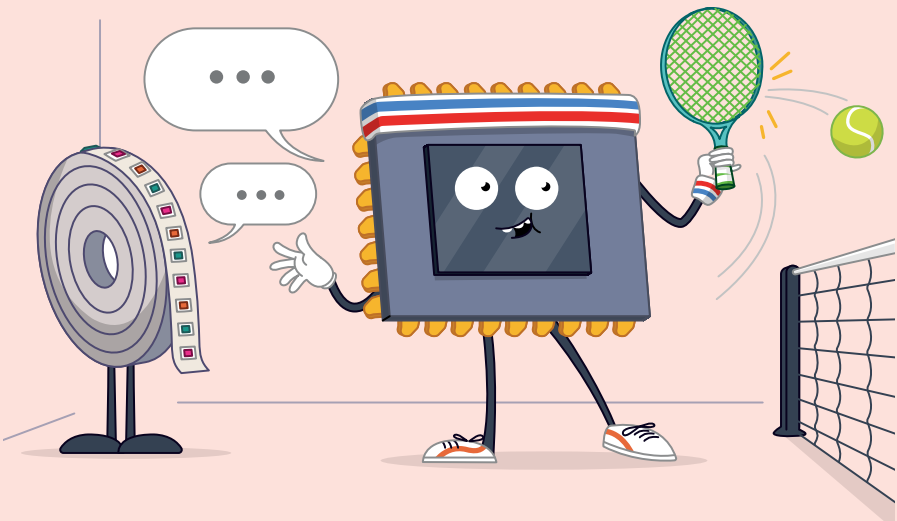
ADC_VREF is the ADC power supply (and reference) voltage, and is generated on Pico by filtering the 3.3 V supply. This pin can be used with an external reference if better ADC performance is required.

AGND is the ground reference for GPIO26–29; there is a separate analogue ground plane running under these signals and terminating at this pin. If the ADC is not used or ADC performance is not critical, this pin can be connected to digital ground.

RUN is the RP2040 enable pin, and has an internal (on-chip) pull-up resistor to 3.3 V of about ~50 kΩ. To reset RP2040, short this pin low.


LED (GP25)
LED
USB
BOOTSEL
DEBUG
Raspberry Pi Pico © 2020
SWCLK
GND
SWDIO
UART0 TX I2C0 SDA SPI0 RX GP0 1
UART0 RX I2C0 SCL SPI0 CSn GP1 2
GND 3
I2C1 SDA SPI0 SCK GP2 4
I2C1 SCL SPI0 TX GP3 5
UART1 TX I2C0 SDA SPI0 RX GP4 6
UART1 RX I2C0 SCL SPI0 CSn GP5 7
GND 8
I2C1 SDA SPI0 SCK GP6 9
I2C1 SCL SPI0 TX GP7 10
UART1 TX I2C0 SDA SPI1 RX GP8 11
UART1 RX I2C0 SCL SPI1 CSn GP9 12
GND 13
I2C1 SDA SPI1 SCK GP10 14
I2C1 SCL SPI1 TX GP11 15
UART0 TX I2C0 SDA SPI1 RX GP12 16
UART0 RX I2C0 SCL SPI1 CSn GP13 17
GND 18
I2C1 SDA SPI1 SCK GP14 19
I2C1 SCL SPI1 TX GP15 20
40 VBUS
39 VSYS
38 GND
37 3V3_EN
36 3V3(OUT)
35 ADC_VREF
34 GP28 ADC2
33 GND AGND
32 GP27 ADC1 I2C1 SCL
31 GP26 ADC0 I2C1 SDA
30 RUN
29 GP22
28 GND
27 GP21 I2C0 SCL
26 GP20 I2C0 SDA
25 GP19 SPI0 TX I2C1 SCL
24 GP18 SPI0 SCK I2C1 SDA
23 GND
22 GP17 SPI0 CSn I2C0 SCL UART0 RX
21 GP16 SPI0 RX I2C0 SDA UART0 TX
Power
Ground
UART / UART (default)
GPIO and PIO
ADC
SPI
I2C
System Control
Debugging

Appendix C
Programmable IO



In this appendix, we look at code that looks very different to code we've dealt with in the rest of the book. That's because we're having to deal with things at a low level. Most of the time, MicroPython can hide a lot of the complexities of how things work on the microcontroller. When we do something like:

```
print("hello")
```

...we don't have to worry about the way the microcontroller stores the letters, or the format in which they get sent to the serial terminal, or the number of clock cycles the serial terminal takes. This is all handled in the background. However, when we get to Programmable Input and Output (PIO), we need to deal with things at a much lower level.

We're going to go on a whistle-stop tour of PIO and introduce some advanced topics so you can get an idea of what's going on, and hopefully understand how PIO on Pico offers some real advantages over the options on other microcontrollers. However, understanding all the low-level data manipulation required to create PIO programs takes time to fully get your head around, so don't worry if it seems a little opaque. If you're interested in tackling this low-level programming, then we'll give you the knowledge to get started and point you in the right direction to continue your journey. If you're more interested in working at a higher level and would rather leave the low-level wrangling to other people, we'll show you how to use PIO programs.

Data in and data out

Throughout this book, we've looked at ways of controlling the pins on your Pico using MicroPython. We can switch them on and off, take inputs, and even send data using the dedicated SPI and I2C controllers. However, what if we want to connect a device that doesn't communicate in SPI or I2C? What if it has its own special protocol?

There are a couple of ways we can do this. On most MicroPython devices, you need to do a process called 'bit banging' where you implement the protocol in MicroPython. Using this, you turn the pins on or off in the right order to send data.

There are three downsides to this. The first is that it's slow. MicroPython does some things really well, but it doesn't run as fast as natively compiled code.

The second is that we have to juggle this with the rest of our code that is running on the microcontroller.

The third is that some timing-critical code can be hard to implement reliably. Fast protocols can need things to happen at very precise times, and with MicroPython we can be quite precise, but if you're trying to transfer megabits a second, you need things to happen every millisecond or possibly every few hundred nanoseconds. That's hard to achieve reliably in MicroPython.

Pico has a solution to this: Programmable IO. There are some extra, really stripped back processing cores that can run simple programs to control the IO pins. You can't program these cores with MicroPython – you have to use a special language just for them – but you can program them from MicroPython. Let's take a look at an example

```
from rp2 import PIO, StateMachine, asm_pio
from machine import Pin
import time

@asm_pio(set_init=PIO.OUT_LOW)
def led_quarter_brightness():
    set(pins, 0) [2]
    set(pins, 1)

@asm_pio(set_init=PIO.OUT_LOW)
def led_half_brightness():
    set(pins, 0)
    set(pins, 1)

@asm_pio(set_init=PIO.OUT_HIGH)
def led_full_brightness():
    set(pins, 1)

sm1 = StateMachine(1, led_quarter_brightness, freq=10000, set_base=Pin(25))
sm2 = StateMachine(2, led_half_brightness, freq=10000, set_base=Pin(25))
```

```
sm3 = StateMachine(3, led_full_brightness, freq=10000, set_base=Pin(25))


while(True):
    sm1.active(1)
    time.sleep(1)
    sm1.active(0)

    sm2.active(1)
    time.sleep(1)
    sm2.active(0)

    sm3.active(1)
    time.sleep(1)
    sm3.active(0)
```

There are three methods here that all look a little strange, but that set the on-board LED to quarter, half, and full brightness. The reason they look a little strange is because they're written in a special language for the PIO system of Pico. You can probably guess what they do – flick the LED on and off very quickly in a similar way to how we used PWM. The instruction **`set(pins, 0)`** turns a GPIO pin off and **`set(pins, 1)`** turns the GPIO pin on.

Each of the three methods has a descriptor above it that tells MicroPython to treat it as a PIO program and not a normal method. These descriptors can also take parameters that influence the behaviour of the programs. In these cases, we've used the **`set_init`** parameter to tell the PIO whether or not the GPIO pin should start off being low or high.

Each of these methods – which are really mini programs that run on the PIO state machines – loops continuously. So, for example, **`led_half_brightness`** will constantly turn the LED on and off so that it spends half its time off and half its time on. **`led_full_brightness`** will similarly loop, but since the only instruction is to turn the LED on, this doesn't actually change anything.

The slightly unusual one here is **`led_quarter_brightness.`** Each PIO instruction takes exactly one clock cycle to run (the length of a clock cycle can be changed by setting the frequency, as we'll see later). However, we can add a number between 1 and 31 in square brackets after an instruction, and this tells the PIO state machine to pause by this number of clock cycles before running the next instruction. In **`led_quarter_brightness,`** then, the two **`set`** instructions each take one clock cycle, and the delay takes two clock cycles, so the total loop takes four clock cycles. In the first line, the **`set`** instruction takes one cycle and the delay takes two, so the GPIO pin is off for three of these four cycles. This makes the LED a quarter as bright as if it were on constantly.

Once you've got your PIO program, you need to load it into a state machine. Since we have three programs, we need to load them into three state machines (there are eight you can use, numbered 0–7). This is done with a line like:

```
sm1 = StateMachine(1, led_quarter_brightness, freq=10000, set_base=Pin(25))
```

The parameters here are:

- The state machine number
- The PIO program to load
- The frequency (which must be between 2000 and 125000000)
- The GPIO pin that the state machine manipulates

There are some additional parameters that you'll see in other programs that we don't need here.

Once you've created your state machine, you can start and stop it using the **`active`** method with 1 (to start) or 0 (to stop). In our loop, we cycle through the three different state machines.

A real example

The previous example was a little contrived, so let's take a look at a way of using PIO with a real example. WS2812B LEDs (sometimes known as NeoPixels) are a type of light that contains three LEDs (one red, one green, and one blue) and a small microcontroller. They're controlled by a single data wire with a timing-dependent protocol that's hard to bit-bang.

Wiring your LED strip is simple, as shown in **Figure C-1**. Depending on the manufacturer of your LED strip, you may have the wires already connected, you may have a socket that you can push header wires in, or you may need to solder them on yourself.

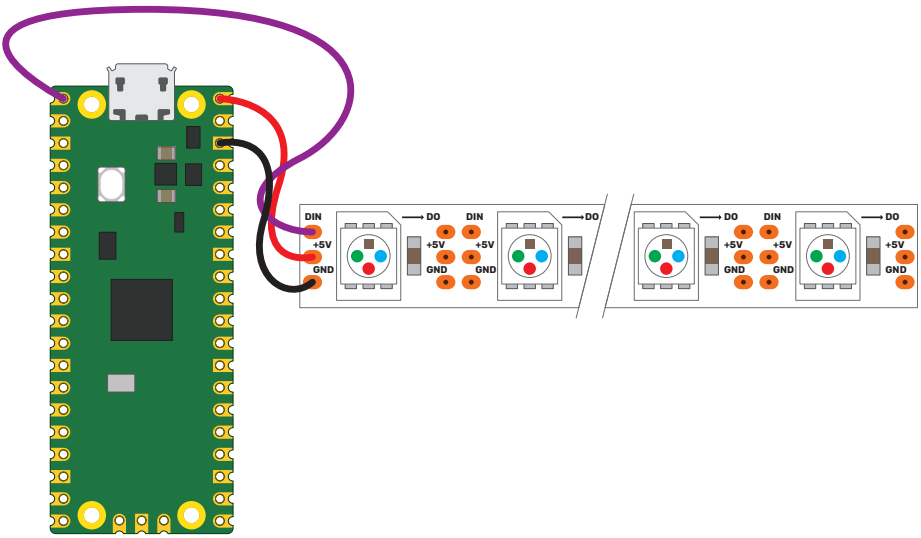


Figure C-1: Connecting an LED strip

One thing you need to be aware of is the potential current draw. While you can add an almost endless series of NeoPixels to your Pico, there's a limit to how much power you can get out

of the 5 V pin on Pico. Here, we'll use eight LEDs, which is perfectly safe, but if you want to use many more than this, you need to understand the limitations and may need to add a separate power supply. You can cut a longer strip to length, and there should be cut lines between the LEDs to show you where to cut. There's a good discussion of the various issues at **hsmag.cc/neopixelpower.**

Now we've got the LEDs wired up, let's take a look at how to control it with PIO:

```
import array, time
from machine import Pin
import rp2
from rp2 import PIO, StateMachine, asm_pio

# Configure the number of WS2812 LEDs.
NUM_LEDS = 10


@asm_pio(sideset_init=PIO.OUT_LOW, out_shiftdir=PIO.SHIFT_LEFT,
autopull=True, pull_thresh=24)
def ws2812():
    T1 = 2
    T2 = 5
    T3 = 3
    label("bitloop")
    out(x, 1)               .side(0)    [T3 - 1]
    jmp(not_x, "do_zero")   .side(1)    [T1 - 1]
    jmp("bitloop")          .side(1)    [T2 - 1]
    label("do_zero")
    nop()                   .side(0)    [T2 - 1]


# Create the StateMachine with the ws2812 program, outputting on Pin(22).
sm = StateMachine(0, ws2812, freq=8000000, sideset_base=Pin(0))

# Start the StateMachine, it will wait for data on its FIFO.
sm.active(1)
```

The basic way that this works is that 800,000 bits of data are sent per second (notice that the frequency is 8000000 and each cycle of the program is 10 clock cycles). Every bit of data is a pulse – a short pulse indicating a 0 and a long pulse indicating a 1. A big difference between this and our previous program is that MicroPython needs to be able to send data to this PIO program.

There are two stages for data coming in to the state machine. The first is a bit of memory called a First In, First Out (or FIFO). This is where our main Python program sends data to. The second is the Output Shift Register (OSR). This is where the **`out()`** instruction fetches data from. The two are linked by **`pull`** instructions which take data from the FIFO and put it in the OSR. However, since our program is set up with **`autopull`** enabled with a threshold of 24, each time we've read 24 bits from the OSR, it will be reloaded from the FIFO.

The instruction **`out(x,1)`** takes one bit of data from the OSR and places it in a variable called **`x`** (there are only two available variables in PIO: **`x`** and **`y`**).

The **`jmp`** instruction tells the code to move directly to a particular label, but it can have a condition. The instruction **`jmp(not_x, "do_zero")`** tells the code to move to **`do_zero`** if the value of x is 0 (or, in logical terms, if **`not_x`** is true, and **`not_x`** is the opposite of **`x`** – in PIO-level speak, 0 is false and any other number is true).

There's a bit of **`jmp`** spaghetti that is mostly there to ensure that the timings are consistent because the loop has to take exactly the same number of cycles every iteration to keep the timing of the protocol in line.

The one aspect we've been ignoring here is the **`.side()`** bits. These are similar to **`set()`** but they take place at the same time as another instruction. This means that **`out(x,1)`** takes place as **`.side(0)`** is setting the value of the sideset pin to 0.

Phew, that's quite a bit going on for such a small program. Now we've got it active, let's look at how to use it. The following code needs to come under the above in your program to send data to a PIO program.

```
# Display a pattern on the LEDs via an array of LED RGB values.
ar = array.array("I", [0 for _ in range(NUM_LEDS)])

print("blue")
for j in range(0, 255):
    for i in range(NUM_LEDS):
        ar[i] = j
    sm.put(ar,8)
    time.sleep_ms(100)

print("red")

for j in range(0, 255):
    for i in range(NUM_LEDS):
        ar[i] = j<<8
    sm.put(ar,8)
    time.sleep_ms(100)
```

```
print("green")
for j in range(0, 255):
    for i in range(NUM_LEDS):
        ar[i] = j<<16
    sm.put(ar,8)
    time.sleep_ms(100)

print("white")
for j in range(0, 255):
    for i in range(NUM_LEDS):
        ar[i] = j<<16 + j<<8 + j
    sm.put(ar,8)
    time.sleep_ms(100)
```

Here we keep track of an array called **ar** that holds the data we want our LEDs to have (we'll look at why we created the array this way in a little while). Each number in the array contains the data for all three colours on a single light. The format is a little strange as it's in binary. One thing about working with PIO is that you often need to work with individual bits of data. Each bit of data is a 1 or 0, and numbers can be built up in this way, so the number 2 in base 10 (as we call normal numbers) is 10 in binary. 3 in base 10 is 11 in binary. The largest number in eight bits of binary is 11111111, or 255 in base 10. We won't go too deep into binary here, but if you want to find out more, you can try the Binary Hero project here: **hsmag.cc/binaryhero**.

To make matters a little more confusing, we're actually storing three numbers in a single number. This is because in MicroPython, whole numbers are stored in 32 bits, but we only need eight bits for each number. There's a little free space at the end as we really only need 24 bits, but that's OK.

The first eight bits are the blue values, the next eight bits are red, and the final eight bits are green. The maximum number you can store in eight bits is 255, so each LED has 255 levels of brightness. We can do this using the bit shift operator <<. This adds a certain number of 0s to the end of a number, so if we want our LED to be at level 1 brightness in red, green, and blue, we start with each value being 1, then shift them the appropriate number of bits. For green, we have:

1 <<16 = 10000000000000000

For red we have:

1 << 8 = 100000000

And for blue, we don't need to shift the bits at all, so we just have 1. If we add all these together, we get the following (if we add the preceding bits to make it a 24-bit number):

```
00000001000000010000001
```

The rightmost eight bits are the blue, the next eight bits are red, and the leftmost eight bits are green.

The final bit that may seem a bit confusing is the line:

```
ar = array.array("I", [0 for _ in range(NUM_LEDS)])
```

This creates an array which has `I` as the first value, and then a 0 for every LED. The reason there's an `I` at the start is that it tells MicroPython that we're using a series of 32-bit values. However, we only want 24 bits of this sent to the PIO for each value, so we tell the `put` command to remove eight bits with:

```
sm.put(ar,8)
```

All the instructions

The language used for PIO state machines is very sparse, so there are only a small number of instructions. In addition to the ones we've looked at, you can use:

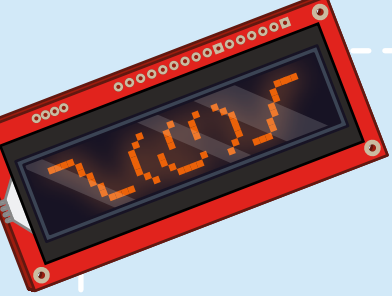
- `in ()` – moves between 1 and 32 bits into the state machine (similar, but opposite to `out()`).
- `push()` – sends data to the memory that links the state machine and the main MicroPython program.
- `pull()` – gets data from the chunk of memory that links the state machine and the main MicroPython program. We haven't used it here because, by including `autopull=True` in our program, this happens automatically when we use `out()`.
- `mov()` – moves data between two locations (such as the `x` and `y` variables).
- `irq()` – controls interrupts. These are used if you need to trigger a particular thing to run on the MicroPython side of your program.
- `wait()` – pauses until something happens (such as a IO pin changes to a set value or an interrupt happens).

WS2812B LIBRARY

While it's useful to experiment with the WS2812B PIO program, if you want to use it in a real project, it may be more useful to use a library that brings it all together. There's one such example at **hsmag.cc/pico-ws2812b**. This lets you create an object that holds all the LED colour data and then use methods such as `set_pixel()` and `fill()` to alter the data. Take a look in the **examples** folder of that repository for more details of how to use it.

Although there are only a small number of possible instructions, it's possible to implement a huge range of communications protocols. Most of the instructions are for moving data about in some form. If you need to prepare the data in any particular way, such as manipulating the colours you want your LEDs to be, this should be done in your main MicroPython program rather than the PIO program.

You can find more information on how to use these, and the full range of options for PIO in MicroPython, on Raspberry Pi Pico in the Pico Python SDK document – and a complete reference to how PIO works in the RP2040 databook. Both of these are available at **rptl.io/rp2040-get-started**.

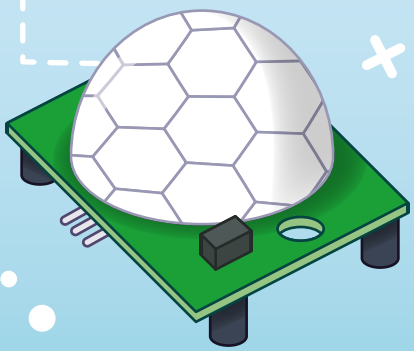


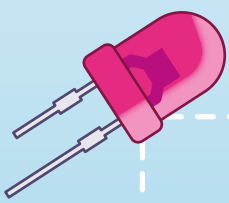
Get started with MicroPython on Raspberry Pi Pico

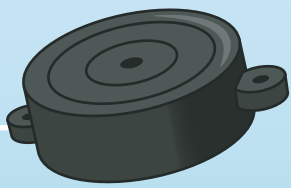
Microcontrollers, like RP2040 at the heart of Raspberry Pi Pico, are computers stripped back to their bare essentials. You don't use monitors or keyboards, but program them to take their input from, and send their output to the input/output pins.

Using these programmable connections, you can light lights, make noises, send text to screens, and much more. In *Get Started with MicroPython on Raspberry Pi Pico*, you will learn how to use the beginner-friendly language MicroPython to write programs and connect up hardware to make your Raspberry Pi Pico interact with the world around it. Using these skills, you can create your own electro-mechanical projects, whether for fun or to make your life easier.

The robotic future is here – you just have to build it yourself. We'll show you how.







raspberrypi.org

Price: £10

9 781912 047864

Raspberry Pi PRESS